Comprendre le langage d’assemblage

Dennis Yurichev

Comprendre le langage d'assemblage

(Rétro-ingénierie pour Débutants)

Pourquoi deux titres? Lire ici: on page xvi.

Dennis Yurichev

my emails

Version du texte (23 octobre 2023).

La dernière version (et l'édition en russe) de ce texte est accessible sur https://beginners.re/.

À la recherche de traducteurs !

Vous souhaitez peut-être m'aider en traduisant ce projet dans d'autres langues, autres que l'anglais et le russe. Il vous suffit de m'envoyer les portions de texte que vous avez traduites (peu importe leur longueur) et je les intégrerai à mon code source LaTeX.

Ne demandez pas si vous pouvez traduire. Faîtes simplement quelque chose. J'ai arrêté de répondre aux courriels "Que puis-je faire".

Lire ici.

Les statistiques par langage sont disponible ici: https://beginners.re/.

La vitesse de traduction n'est pas importante, puisqu'il s'agit d'un projet open-source après tout. Votre nom sera mentionné en tant que contributeur au projet. Les traductions en coréen, chinois et persan sont réservées par mes éditeurs.

Les versions anglaise et russe ont été réalisées par moi-même. Toutefois, mon anglais est toujours horrible et je vous serais très reconnaissant pour toute éventuelle remarque sur la grammaire, etc... Même mon russe est imparfait, donc je vous serais également reconnaissant pour toute remarque sur la traduction en russe !

N'hésitez donc pas à me contacter : my emails.

Contenus abrégés

Table des matières

[1]Floating-Point Unit

[2]Java Virtual Machine
[3]Nombre généré pseudo-aléatoirement

[4] Instruction Set Architecture

[5] Initialization Vector

Préface

C'est quoi ces deux titres?

Le livre a été appelé "Reverse Engineering for Beginners" en 2014-2018, mais j'ai toujours suspecté que ça rendait son audience trop réduite.

Les gens de l'infosec connaissent le "reverse engineering", mais j'ai rarement entendu le mot "assembleur" de leur part.

De même, le terme "reverse engineering" est quelque peu cryptique pour une audience générale de programmeurs, mais qui ont des connaissances à propos de l'"assembleur".

En juillet 2018, à titre d'expérience, j'ai changé le titre en "Assembly Language for Beginners" et publié le lien sur le site Hacker News[6], et le livre a été plutôt bien accueilli.

Donc, c'est ainsi que le livre a maintenant deux titres.

Toutefois, j'ai changé le second titre à "Understanding Assembly Language", car quelqu'un a déjà écrit le livre "Assembly Language for Beginners". De même, des gens disent que "for Beginners" sonne sarcastique pour un livre de ~1000 pages.

Les deux livres diffèrent seulement par le titre, le nom du fichier (UAL-XX.pdf versus RE4B-XX.pdf), l'URL et quelques-une des première pages.

À propos de la rétro-ingénierie

Il existe plusieurs définitions pour l'expression «ingénierie inverse ou rétro-ingénierie reverse engineering» :

1) L'ingénierie inverse de logiciels : examiner des programmes compilés;

2) Le balayage des structures en 3D et la manipulation numérique nécessaire afin de les reproduire;

3) Recréer une structure de base de données.

Ce livre concerne la première définition.

Prérequis

Connaissance basique du C LP[7]. Il est recommandé de lire: 12.1.3 on page 1315.

Exercices et tâches

…ont été déplacés sur un site différent : http://challenges.re.

[6] https://news.ycombinator.com/item?id=17549050

[7] Langage de programmation

Éloges de ce livre

https://beginners.re/#praise.

Universités

Ce livre est recommandé par au moins ces universités: https://beginners.re/#uni.

Remerciements

Pour avoir patiemment répondu à toutes mes questions : SkullC0DEr.

Pour m'avoir fait des remarques par rapport à mes erreurs ou manques de précision : Alexander Lysenko, Federico Ramondino, Mark Wilson, Razikhova Meiramgul Kayratovna, Anatoly Prokofiev, Kostya Begunets, Valentin "netch" Nechayev, Aleksandr Plakhov, Artem Metla, Alexander Yastrebov, Vlad Golovkin[8], Evgeny Proshin, Alexander Myasnikov, Alexey Tretiakov, Oleg Peskov, Pavel Shakhov, Zhu Ruijin, Changmin Heo, Vitor Vidal, Stijn Crevits, Jean-Gregoire Foulon[9], Ben L., Etienne Khan, Norbert Szetei[10], Marc Remy, Michael Hansen, Derk Barten, The Renaissance[11], Hugo Chan, Emil Mursalimov, Tanner Hoke, Tan90909090@GitHub, Ole Petter Orhagen, Sourav Punoriyar, Vitor Oliveira, Alexis Ehret, Maxim Shlochiski, Greg Paton, Pierrick Lebourgeois, Abdullah Alomair, Bobby Battista, Ashod Nakashian..

Pour m'avoir aidé de toute autre manière : Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), noshadow on #gcc IRC, Aliaksandr Autayeu, Mohsen Mostafa Jokar, Peter Sovietov, Misha "tiphareth" Verbitsky.

Pour avoir traduit le livre en chinois simplifié : Antiy Labs (antiy.cn), Archer.

Pour avoir traduit le livre en coréen : Byungho Min.

Pour avoir traduit le livre en néerlandais : Cedric Sambre (AKA Midas).

Pour avoir traduit le livre en espagnol : Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida, Diogo Mussi, Patricio Galdames, Emiliano Estevarena.

Pour avoir traduit le livre en portugais : Thales Stevan de A. Gois, Diogo Mussi, Luiz Filipe, Primo David Santini.

Pour avoir traduit le livre en italien : Federico Ramondino[12], Paolo Stivanin[13], twyK, Fabrizio Bertone, Matteo Sticco, Marco Negro[14], bluepulsar.

Pour avoir traduit le livre en français : Florent Besnard[15], Marc Remy[16], Baudouin

[8] goto-vlad@github
[9] https://github.com/pixjuan
[10] https://github.com/73696e65
[11] https://github.com/TheRenaissance
[12] https://github.com/pinkrab
[13] https://github.com/paolostivanin
[14] https://github.com/Internaut401
[15] https://github.com/besnardf
[16] https://github.com/mremy

Landais, Téo Dacquet[17], BlueSkeye@GitHub[18].

Pour avoir traduit le livre en allemand : Dennis Siekmeier[19], Julius Angres[20], Dirk Loser[21], Clemens Tamme, Philipp Schweinzer, Tobias Deiminger.

Pour avoir traduit le livre en polonais: Kateryna Rozanova, Aleksander Mistewicz, Wiktoria Lewicka, Marcin Sokołowski.

Pour avoir traduit le livre en japonais: shmz@github[22],4ryuJP@github[23].

Pour la relecture : Vladimir Botov, Andrei Brazhuk, Mark "Logxen" Cooper, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen, Hong Xie.

Vasil Kolev[24] a réalisé un gros travail de relecture et a corrigé beaucoup d'erreurs.

Merci également à toutes les personnes sur github.com qui ont contribué aux remarques et aux corrections.

De nombreux packages LaTeX ont été utilisé : j'aimerais également remercier leurs auteurs.

Donateurs

Ceux qui m'ont soutenu lorsque j'écrivais le livre :

2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar ($50), James Truscott ($4.5), Luis Rocha ($63), Joris van de Vis ($127), Richard S Shultz ($20), Jang Minchang ($20), Shade Atlas (5 AUD), Yao Xiao ($10), Pawel Szczur (40 CHF), Justin Simms ($20), Shawn the R0ck ($27), Ki Chan Ahn ($50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararao ($50), Philippe Teuwen ($4), Martin Haeberli ($10), Victor Cazacov (€5), Tobias Sturzenegger (10 CHF), Sonny Thai ($15), Bayna AlZaabi ($75), Redfive B.V. (€25), Joona Oskari Heikkilä (€5), Marshall Bishop ($50), Nicolas Werner (€12), Jeremy Brown ($100), Alexandre Borges ($25), Vladimir Dikovski (€50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent ($30), Sri Harsha Kandrakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe ($14), Maxim Dyakonov ($3), Sebastian Aguilera (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov ($100), Yuri Romanov (1000 RUR), Aliaksandr Autayeu (€10), Tudor Azoitei ($40), Z0vsky (€10), Yu Dai ($10), Anonymous ($15), Vladislav Chelnokov ($25), Nenad Noveljic ($50), Ryan Smith ($25), Andreas Schommer (€5), Nikolay Gavrilov ($300), Ernesto Bonev Reynoso ($30).

Un énorme merci à chaque donateur !

[17] https://github.com/T30rix
[18] https://github.com/BlueSkeye
[19] https://github.com/DSiekmeier
[20] https://github.com/JAngres
[21] https://github.com/PolymathMonkey
[22] https://github.com/shmz
[23] https://github.com/4ryuJP
[24] https://vasil.ludost.net/

mini-FAQ

Q: Est-ce que ce livre est plus simple/facile que les autres?

R: Non, c'est à peu près le même niveau que les autres livres sur ce sujet.

Q: J'ai trop peur de commencer à lire ce livre, il fait plus de 1000 pages. "...for Beginners" dans le nom sonne un peu sarcastique.

R: Toutes sortes de listings constituent le gros de ce livre. Le livre est en effet pour les débutants, il manque (encore) beaucoup de choses.

Q: Quels sont les pré-requis nécessaires avant de lire ce livre ?

R: Une compréhension de base du C/C++ serait l'idéal.

Q: Dois-je apprendre x86/x64/ARM et MIPS en même temps ? N'est-ce pas un peu trop ?

R: Je pense que les débutants peuvent seulement lire les parties x86/x64, tout en passant/feuilletant celles ARM/MIPS.

Q: Puis-je acheter une version papier du livre en russe / anglais ?

R: Malheureusement non, aucune maison d'édition n'a été intéressée pour publier une version en russe ou en anglais du livre jusqu'à présent. Cependant, vous pouvez demander à votre imprimerie préférée de l'imprimer et de le relier. `https://yurichev.com/news/20200222_printed_RE4B/`.

Q: Y a-il une version ePub/Mobi ?

R: Le livre dépend majoritairement de TeX/LaTeX, il n'est donc pas évident de le convertir en version ePub/Mobi.

Q: Pourquoi devrait-on apprendre l'assembleur de nos jours ?

R: A moins d'être un développeur d'OS[25], vous n'aurez probablement pas besoin d'écrire en assembleur—les derniers compilateurs (ceux de notre décennie) sont meilleurs que les êtres humains en terme d'optimisation. [26].

De plus, les derniers CPU[27]s sont des appareils complexes et la connaissance de l'assembleur n'aide pas vraiment à comprendre leurs mécanismes internes.

Cela dit, il existe au moins deux domaines dans lesquels une bonne connaissance de l'assembleur peut être utile : Tout d'abord, pour de la recherche en sécurité ou sur des malwares. C'est également un bon moyen de comprendre un code compilé lorsqu'on le debug. Ce livre est donc destiné à ceux qui veulent comprendre l'assembleur plutôt que d'écrire en assembleur, ce qui explique pourquoi il y a de nombreux exemples de résultats issus de compilateurs dans ce livre.

Q: J'ai cliqué sur un lien dans le document PDF, comment puis-je retourner en arrière ?

R: Dans Adobe Acrobat Reader, appuyez sur Alt + Flèche gauche. Dans Evince, appuyez sur le bouton "<".

[25] Système d'exploitation (Operating System)

[26] Un très bon article à ce sujet : [Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)]

[27] Central Processing Unit

Q: Puis-je imprimer ce livre / l'utiliser pour de l'enseignement ?

R: Bien sûr ! C'est la raison pour laquelle le livre est sous licence Creative Commons (CC BY-SA 4.0).

Q: Pourquoi ce livre est-il gratuit ? Vous avez fait du bon boulot. C'est suspect, comme nombre de choses gratuites.

R: D'après ma propre expérience, les auteurs d'ouvrages techniques font cela pour l'auto-publicité. Il n'est pas possible de se faire beaucoup d'argent d'une telle manière.

Q: Comment trouver du travail dans le domaine de la rétro-ingénierie ?

R: Il existe des sujets d'embauche qui apparaissent de temps en temps sur Reddit, dédiés à la rétro-ingénierie (cf. reverse engineering ou RE)[28]. Jetez un œil ici.

Un sujet d'embauche quelque peu lié peut être trouvé dans le subreddit «netsec».

Q: J'ai une question...

R: Envoyez-la moi par email (my emails).

À propos de la traduction en Coréen

En Janvier 2015, la maison d'édition Acorn (www.acornpub.co.kr) en Corée du Sud a réalisé un énorme travail en traduisant et en publiant mon livre (dans son état en Août 2014) en Coréen.

Il est désormais disponible sur leur site web.

Le traducteur est Byungho Min (twitter/tais9). L'illustration de couverture a été réalisée l'artiste, Andy Nechaevsky, un ami de l'auteur: facebook/andydinka. Ils détiennent également les droits d'auteurs sur la traduction coréenne.

Donc si vous souhaitez avoir un livre *réel* en coréen sur votre étagère et que vous souhaitez soutenir ce travail, il est désormais disponible à l'achat.

Á propos de la traduction en Farsi/Perse

En 2016, ce livre a été traduit par Mohsen Mostafa Jokar (qui est aussi connu dans la communauté iranienne pour sa traduction du manuel de Radare[29]). Il est disponible sur le site web de l'éditeur[30] (Pendare Pars).

Extrait de 40 pages: `https://beginners.re/farsi.pdf`.

Enregistrement du livre à la Bibliothèque Nationale d'Iran: `http://opac.nlai.ir/opac-prod/bibliographic/4473995`.

[28] reddit.com/r/ReverseEngineering/
[29] `http://rada.re/get/radare2book-persian.pdf`
[30] `http://goo.gl/2Tzx0H`

Á propos de la traduction en Chinois

En avril 2017, la traduction en Chinois a été terminée par Chinese PTPress. Ils sont également les détenteurs des droits de la traduction en Chinois.

La version chinoise est disponible à l'achat ici: `http://www.epubit.com.cn/book/details/4174`. Une revue partielle et l'historique de la traduction peut être trouvé ici: `http://www.cptoday.cn/news/detail/3155`.

Le traducteur principal est Archer, à qui je dois beaucoup. Il a été très méticuleux (dans le bon sens du terme) et a signalé la plupart des erreurs et bugs connus, ce qui est très important dans le genre de littérature de ce livre. Je recommanderais ses services à tout autre auteur!

Les gens de Antiy Labs ont aussi aidé pour la traduction. Voici la préface écrite par eux.

Chapitre 1

Pattern de code

1.1 La méthode

Lorsque j'ai commencé à apprendre le C, et plus tard, le C++, j'ai pris l'habitude d'écrire des petits morceaux de code, de les compiler et de regarder le langage d'assemblage généré. Cela m'a permis de comprendre facilement ce qui se passe dans le code que j'écris. [1]. Je l'ai fait si souvent que la relation entre le code C++ et ce que le compilateur produit a été imprimée profondément dans mon esprit. Ça m'est facile d'imaginer immédiatement l'allure de la fonction et du code C. Peut-être que cette méthode pourrait être utile à d'autres.

Parfois, des anciens compilateurs sont utilisés, afin d'obtenir des extraits de code le plus court (ou le plus simple) possible.

À propos, il y a un bon site où vous pouvez faire la même chose, avec de nombreux compilateurs, au lieu de les installer sur votre système. Vous pouvez également l'utiliser: `https://godbolt.org/`.

Exercices

Lorsque j'étudiais le langage d'assemblage, j'ai souvent compilé des petites fonctions en C et les ai ensuite récrites peu à peu en assembleur, en essayant d'obtenir un code aussi concis que possible. Cela n'en vaut probablement plus la peine aujourd'hui, car il est difficile de se mesurer aux derniers compilateurs en terme d'efficacité. Cela reste par contre un excellent moyen d'approfondir ses connaissances de l'assembleur. N'hésitez pas à prendre n'importe quel code assembleur de ce livre et à essayer de le rendre plus court. Toutefois, n'oubliez pas de tester ce que vous aurez écrit.

[1] En fait, je le fais encore cela lorsque je ne comprends pas ce qu'un morceau de code fait. Exemple récent de 2019: `p += p+(i&1)+2;` tiré de "SAT0W" solveur SAT par D.Knuth.

Niveau d'optimisation et information de débogage

Le code source peut être compilé par différents compilateurs, avec des niveaux d'optimisation variés. Un compilateur en a typiquement trois, où le niveau 0 désactive l'optimisation. L'optimisation peut se faire en ciblant la taille du code ou la vitesse d'exécution. Un compilateur sans optimisation est plus rapide et produit un code plus compréhensible (quoique verbeux), alors qu'un compilateur avec optimisation est plus lent et essaye de produire un code qui s'exécute plus vite (mais pas forcément plus compact). En plus des niveaux d'optimisation, un compilateur peut inclure dans le fichier généré des informations de débogage, qui produit un code facilitant le débogage. Une des caractéristiques importante du code de 'debug', est qu'il peut contenir des liens entre chaque ligne du code source et les adresses du code machine associé. D'un autre côté, l'optimisation des compilateurs tend à générer du code où des lignes du code source sont modifiées, et même parfois absentes du code machine résultant. Les rétro-ingénieurs peuvent rencontrer n'importe quelle version, simplement parce que certains développeurs mettent les options d'optimisation, et d'autres pas. Pour cette raison, et lorsque c'est possible, nous allons essayer de travailler sur des exemples avec les versions de débogage et finale du code présenté dans ce livre.

1.2 Quelques bases

1.2.1 Une courte introduction sur le CPU

Le CPU est le système qui exécute le code machine que constitue le programme.

Un court glossaire:

Instruction : Une commande CPU primitive. Les exemples les plus simples incluent: déplacement de données entre les registres, travail avec la mémoire et les opérations arithmétiques primitives. Généralement, chaque CPU a son propre jeu d'instructions (ISA).

Code machine : Code que le CPU exécute directement. Chaque instruction est codée sur plusieurs octets.

Langage d'assemblage : Code mnémotechnique et quelques extensions comme les macros qui facilitent la vie du programmeur.

Registre CPU : Chaque CPU a un ensemble de registres d'intérêt général (GPR[2]). ≈ 8 pour x86, ≈ 16 pour x86-64, ≈ 16 pour ARM. Le moyen le plus simple de comprendre un registre est de le voir comme une variable temporaire non-typée. Imaginez que vous travaillez avec un LP de haut niveau et que vous pouvez utiliser uniquement huit variables de 32-bit (ou de 64-bit). C'est malgré tout possible de faire beaucoup de choses en les utilisant!

On pourrait se demander pourquoi il y a besoin d'une différence entre le code machine et un LP. La réponse est que les humains et les CPUs ne sont pas semblables—c'est beaucoup plus simple pour les humains d'utiliser un LP de haut niveau comme

[2]General Purpose Registers

C/C++, Java, Python, etc., mais c'est plus simple pour un CPU d'utiliser un niveau d'abstraction de beaucoup plus bas niveau. Peut-être qu'il serait possible d'inventer un CPU qui puisse exécuter du code d'un LP de haut niveau, mais il serait beaucoup plus complexe que les CPUs que nous connaissons aujourd'hui. D'une manière similaire, c'est moins facile pour les humains d'écrire en langage d'assemblage à cause de son bas niveau et de la difficulté d'écrire sans faire un nombre énorme de fautes agaçantes. Le programme qui convertit d'un LP haut niveau vers l'assemblage est appelé un *compilateur*.

Quelques mots sur les différents ISAs

Le jeu d'instructions ISA x86 a toujours été avec des instructions de taille variable. Donc quand l'époque du 64-bit arriva, les extensions x64 n'ont pas impacté le ISA très significativement. En fait, le ISA x86 contient toujours beaucoup d'instructions apparues pour la première fois dans un CPU 8086 16-bit, et que l'on trouve encore dans beaucoup de CPUs aujourd'hui. ARM est un CPU RISC[3] conçu avec l'idée d'instructions de taille fixe, ce qui présentait quelques avantages dans le passé. Au tout début, toutes les instructions ARM étaient codés sur 4 octets[4]. C'est maintenant connu comme le «ARM mode». Ensuite ils sont arrivés à la conclusion que ce n'était pas aussi économique qu'ils l'avaient imaginé sur le principe. En réalité, la majorité des instructions CPU utilisées[5] dans le monde réel peuvent être encodées en utilisant moins d'informations. Ils ont par conséquent ajouté un autre ISA, appelé Thumb, où chaque instruction était encodée sur seulement 2 octets. C'est maintenant connu comme le «Thumb mode». Cependant, toutes les instructions ne peuvent être encodées sur seulement 2 octets, donc les instructions Thumb sont un peu limitées. On peut noter que le code compilé pour le mode ARM et pour le mode Thumb peut, évidemment, coexister dans un seul programme. Les créateurs de ARM pensèrent que Thumb pourrait être étendu, donnant naissance à Thumb-2, qui apparut dans ARMv7. Thumb-2 utilise toujours des instructions de 2 octets, mais a de nouvelles instructions dont la taille est de 4 octets. Une erreur couramment répandue est que Thumb-2 est un mélange de ARM et Thumb. C'est incorrect. Plutôt, Thumb-2 fut étendu pour supporter totalement toutes les caractéristiques du processeur afin qu'il puisse rivaliser avec le mode ARM—un objectif qui a été clairement réussi, puisque la majorité des applications pour iPod/iPhone/iPad est compilée pour le jeu d'instructions de Thumb-2 (il est vrai que c'est largement dû au fait que Xcode le faisait par défaut). Plus tard, le ARM 64-bit sortit. Ce ISA a des instructions de 4 octets, et enlevait le besoin d'un mode Thumb supplémentaire. Cependant, les prérequis de 64-bit affectèrent le ISA, résultant maintenant au fait que nous avons trois jeux d'instructions ARM: ARM mode, Thumb mode (incluant Thumb-2) et ARM64. Ces ISAs s'intersectent partiellement, mais on peut dire que ce sont des ISAs différents, plutôt que des variantes du même. Par conséquent, nous essayerons d'ajouter des fragments de code dans les trois ISAs de ARM dans ce livre. Il y a, d'ailleurs, bien d'autres ISAs RISC avec des instructions de taille fixe de 32-bit, comme MIPS, PowerPC et Alpha AXP.

[3]Reduced Instruction Set Computing

[4]D'ailleurs, les instructions de taille fixe sont pratiques parce qu'il est possible de calculer l'instruction suivante (ou précédente) sans effort. Cette caractéristique sera discutée dans la section de l'opérateur switch() (1.21.2 on page 227).

[5]Ce sont MOV/PUSH/CALL/Jcc

1.2.2 Systèmes de numération

> Nowadays octal numbers seem to be used for exactly one purpose—file permissions on POSIX systems—but hexadecimal numbers are widely used to emphasize the bit pattern of a number over its numeric value.
>
> Alan A. A. Donovan, Brian W. Kernighan — The Go Programming Language

Les Hommes ont probablement pris l'habitude d'utiliser la numérotation décimale parce qu'ils ont 10 doigts. Néanmoins, le nombre 10 n'a pas de signification particulière en science et en mathématiques. En électronique, le système de numérotation est le binaire : 0 pour l'absence de courant dans un fil et 1 s'il y en a. 10 en binaire est 2 en décimal; 100 en binaire est 4 en décimal et ainsi de suite.

Si le système de numération a 10 chiffres, il est en *base* 10. Le système binaire est en *base* 2.

Choses importantes à retenir:

1) Un *nombre* est un nombre, tandis qu'un *chiffre* est un élément d'un système d'écriture et est généralement un caractère

2) Un nombre ne change pas lorsqu'on le convertit dans une autre base; seule sa représentation écrite change (et donc la façon de le représenter en RAM[6]).

1.2.3 Conversion d'une base à une autre

La notation positionnelle est utilisée dans presque tous les systèmes de numération, cela signifie qu'un chiffre a un poids dépendant de sa position dans la représentation du nombre. Si 2 se situe le plus à droite, c'est 2. S'il est placé un chiffre avant celui le plus à droite, c'est 20.

Que représente 1234 ?

$10^3 \cdot 1 + 10^2 \cdot 2 + 10^1 \cdot 3 + 1 \cdot 4$ = 1234 ou $1000 \cdot 1 + 100 \cdot 2 + 10 \cdot 3 + 4 = 1234$

De même pour les nombres binaires, mais la base est 2 au lieu de 10. Que représente 0b101011 ?

$2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 43$ ou $32 \cdot 1 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 = 43$

Il existe aussi la notation non-positionnelle comme la numération romaine[7]. Peut-être que l'humanité a choisi le système de numération positionnelle parce qu'il était plus simple pour les opérations basiques (addition, multiplication, etc.) à la main sur papier.

En effet, les nombres binaires peuvent être ajoutés, soustraits et ainsi de suite de la même manière que c'est enseigné à l'école, mais seulement 2 chiffres sont disponibles.

[6] Random-Access Memory

[7] À propos de l'évolution du système de numération, voir [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 195–213.]

Les nombres binaires sont volumineux lorsqu'ils sont représentés dans le code source et les dumps, c'est pourquoi le système hexadécimal peut être utilisé. La base hexadécimale utilise les nombres 0..9 et aussi 6 caractères latins : A..F. Chaque chiffre hexadécimal prend 4 bits ou 4 chiffres binaires, donc c'est très simple de convertir un nombre binaire vers l'hexadécimal et inversement, même manuellement, de tête.

hexadécimal	binaire	décimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Comment savoir quelle est la base utilisée dans un cas particulier?

Les nombres décimaux sont d'ordinaire écrits tels quels, i.e, 1234. Certains assembleurs permettent d'accentuer la base décimale, et les nombres peuvent alors s'écrire avec le suffixe "d" : 1234d.

Les nombres binaires sont parfois préfixés avec "0b" : 0b100110111 (GCC[8] a une extension de langage non-standard pour ça [9]). Il y a aussi un autre moyen : le suffixe "b", par exemple : 100110111b. J'essaierai de garder le préfixe "0b" tout le long du livre pour les nombres binaires.

Les nombres hexadécimaux sont préfixés avec "0x" en C/C++ et autres LPs : 0x1234ABCD. Ou ils ont le suffixe "h" : 1234ABCDh - c'est une manière commune de les représenter dans les assembleurs et les débogueurs. Si le nombre commence par un A..F, un 0 est ajouté au début : 0ABCDEFh. Il y avait une convention répandue à l'ère des ordinateurs personnels 8-bit, utilisant le préfixe $, comme $ABCD. J'essaierai de garder le préfixe "0x" tout le long du livre pour les nombres hexadécimaux.

Faut-il apprendre à convertir les nombres de tête? La table des nombres hexadécimaux de 1 chiffre peut facilement être mémorisée. Pour les nombres plus gros, ce n'est pas la peine de se tourmenter.

[8] GNU Compiler Collection
[9] https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html

Peut-être que les nombres hexadécimaux les plus visibles sont dans les URL[10]s. C'est la façon d'encoder les caractères non-Latin. Par exemple: `https://en.wiktionary.org/wiki/na%C3%AFvet%C3%A9` est l'URL de l'article de Wiktionary à propos du mot «naïveté».

Base octale

Un autre système de numération a été largement utilisé en informatique est la représentation octale. Elle comprend 8 chiffres (0..7), et chacun occupe 3 bits, donc c'est facile de convertir un nombre d'une base à l'autre. Il est maintenant remplacé par le système hexadécimal quasiment partout mais, chose surprenante, il y a encore une commande sur *NIX, utilisée par beaucoup de personnes, qui a un nombre octal comme argument : `chmod`.

Comme beaucoup d'utilisateurs *NIX le savent, l'argument de `chmod` peut être un nombre à 3 chiffres. Le premier correspond aux droits du propriétaire du fichier, le second correspond aux droits pour le groupe (auquel le fichier appartient), le troisième est pour tous les autres. Et chaque chiffre peut être représenté en binaire:

décimal	binaire	signification
7	111	**rwx**
6	110	**rw-**
5	101	**r-x**
4	100	**r--**
3	011	**-wx**
2	010	**-w-**
1	001	**--x**
0	000	**---**

Ainsi chaque bit correspond à un droit: lecture (r) / écriture (w) / exécution (x).

L'importance de `chmod` est que le nombre entier en argument peut être écrit comme un nombre octal. Prenons par exemple, 644. Quand vous tapez `chmod 644 file`, vous définissez les droits de lecture/écriture pour le propriétaire, les droits de lecture pour le groupe et encore les droits de lecture pour tous les autres. Convertissons le nombre octal 644 en binaire, ça donne `110100100`, ou (par groupe de 3 bits) `110 100 100`.

Maintenant que nous savons que chaque triplet sert à décrire les permissions pour le propriétaire/groupe/autres : le premier est `rw-`, le second est `r--` et le troisième est `r--`.

Le système de numération octal était aussi populaire sur les vieux ordinateurs comme le PDP-8 parce que les mots pouvaient être de 12, 24 ou de 36 bits et ces nombres sont divisibles par 3, donc la représentation octale était naturelle dans cet environnement. Aujourd'hui, tous les ordinateurs populaires utilisent des mots/taille d'adresse de 16, 32 ou de 64 bits et ces nombres sont divisibles par 4, donc la représentation hexadécimale était plus naturelle ici.

[10]Uniform Resource Locator

Le système de numération octal est supporté par tous les compilateurs C/C++ standards. C'est parfois une source de confusion parce que les nombres octaux sont notés avec un zéro au début. Par exemple, 0377 est 255. Et parfois, vous faites une faute de frappe et écrivez ”09” au lieu de 9, et le compilateur renvoie une erreur. GCC peut renvoyer quelque chose comme ça:

```
erreur: chiffre 9 invalide dans la constante en base 8.
```

De même, le système octal est assez populaire en Java. Lorsque IDA[11] affiche des chaînes Java avec des caractères non-imprimables, ils sont encodés dans le système octal au lieu d'hexadécimal. Le décompilateur Java JAD se comporte de la même façon.

Divisibilité

Quand vous voyez un nombre décimal comme 120, vous en déduisez immédiatement qu'il est divisible par 10, parce que le dernier chiffre est zéro. De la même façon, 123400 est divisible par 100 parce que les deux derniers chiffres sont zéros.

Pareillement, le nombre hexadécimal 0x1230 est divisible par 0x10 (ou 16), 0x123000 est divisible par 0x1000 (ou 4096), etc.

Un nombre binaire 0b1000101000 est divisible par 0b1000 (8), etc.

Cette propriété peut être souvent utilisée pour déterminer rapidement si l'adresse ou la taille d'un bloc mémoire correspond à une limite. Par exemple, les sections dans les fichiers PE[12] commencent quasiment toujours à une adresse finissant par 3 zéros hexadécimaux: 0x41000, 0x10001000, etc. La raison sous-jacente est que la plupart des sections PE sont alignées sur une limite de 0x1000 (4096) octets.

Arithmétique multi-précision et base

L'arithmétique multi-précision utilise des nombres très grands et chacun peut être stocké sur plusieurs octets. Par exemple, les clés RSA, tant publique que privée, utilisent jusqu'à 4096 bits et parfois plus encore.

Dans [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 265] nous trouvons l'idée suivante: quand vous stockez un nombre multi-précision dans plusieurs octets, le nombre complet peut être représenté dans une base de $2^8 = 256$, et chacun des chiffres correspond à un octet. De la même manière, si vous sauvegardez un nombre multi-précision sur plusieurs entiers de 32 bits, chaque chiffre est associé à l'emplacement de 32 bits et vous pouvez penser à ce nombre comme étant stocké dans une base 2^{32}.

Comment prononcer les nombres non-décimaux

Les nombres dans une base non décimale sont généralement prononcés un chiffre à la fois : “un-zéro-zéro-un-un-...”. Les mots comme “dix“, “mille“, etc, ne sont généralement pas prononcés, pour éviter d'être confondus avec ceux en base décimale.

[11] Désassembleur interactif et débogueur développé par Hex-Rays
[12] Portable Executable

Nombres à virgule flottante

Pour distinguer les nombres à virgule flottante des entiers, ils sont souvent écrits avec avec un “.0“ à la fin, comme 0.0, 123.0, etc.

1.3 Fonction vide

La fonction la plus simple possible est sans doute celle qui ne fait rien:

Listing 1.1 : Code C/C++

```
void f()
{
        return;
};
```

Compilons-la!

1.3.1 x86

Voici ce que les compilateurs GCC et MSVC produisent sur une plateforme x86:

Listing 1.2 : GCC/MSVC avec optimisation (résultat en sortie de l’assembleur)

```
f:
        ret
```

Il y a juste une instruction: RET, qui détourne l’exécution vers l’appelant.

1.3.2 ARM

Listing 1.3 : avec optimisation Keil 6/2013 (Mode ARM) ASM Output

```
f       PROC
        BX       lr
        ENDP
```

L’adresse de retour n’est pas stockée sur la pile locale avec l’ISA ARM, mais dans le ”link register” (registre de lien), donc l’instruction BX LR force le flux d’exécution à sauter à cette adresse—renvoyant effectivement l’exécution vers l’appelant.

1.3.3 MIPS

Il y a deux conventions de nommage utilisées dans le monde MIPS pour nommer les registres: par numéro (de $0 à $31) ou par un pseudo-nom ($V0, $A0, etc.).

La sortie de l’assembleur GCC ci-dessous liste les registres par numéro:

Listing 1.4 : GCC 4.4.5 avec optimisation (résultat en sortie de l’assembleur)

```
        j       $31
        nop
```

…tandis qu'IDA le fait—avec les pseudo noms:

Listing 1.5 : GCC 4.4.5 avec optimisation (IDA)

```
        j       $ra
        nop
```

La première instruction est l'instruction de saut (J ou JR) qui détourne le flux d'exécution vers l'appelant, sautant à l'adresse dans le registre $31 (ou $RA).

Ce registre est similaire à LR[13] en ARM.

La seconde instruction est NOP[14], qui ne fait rien. Nous pouvons l'ignorer pour l'instant.

Une note à propos des instructions MIPS et des noms de registres

Les registres et les noms des instructions dans le monde de MIPS sont traditionnellement écrits en minuscules. Cependant, dans un souci d'homogénéité, nous allons continuer d'utiliser les lettres majuscules, étant donné que c'est la convention suivie par tous les autres ISAs présentés dans ce livre.

1.3.4 Fonctions vides en pratique

Bien que les fonctions vides soient inutiles, elles sont assez fréquentes dans le code bas niveau.

Tout d'abord, les fonctions de débogage sont assez populaires, comme celle-ci:

Listing 1.6 : Code C/C++

```
void dbg_print (const char *fmt, ...)
{
#ifdef _DEBUG
        // open log file
        // write to log file
        // close log file
#endif
};

void some_function()
{
        ...

        dbg_print ("we did something\n");

        ...
};
```

Dans une compilation en non-debug (e.g., “release”), `_DEBUG` n'est pas défini, donc la fonction `dbg_print()`, bien qu'elle soit appelée pendant l'exécution, sera vide.

[13] Link Register
[14] No Operation

Un autre moyen de protection logicielle est de faire plusieurs compilations: une pour les clients, une de démonstration. La compilation de démonstration peut omettre certaines fonctions importantes, comme ici:

Listing 1.7 : Code C/C++

```
void save_file ()
{
#ifndef DEMO
        // a real saving code
#endif
};
```

La fonction `save_file()` peut être appelée lorsque l'utilisateur clique sur le menu `Fichier->Enregistrer`. La version de démo peut être livrée avec cet item du menu désactivé, mais même si un logiciel cracker pourra l'activer, une fonction vide sans code utile sera appelée.

IDA signale de telles fonctions avec des noms comme `nullsub_00`, `nullsub_01`, etc.

1.4 Valeur de retour

Une autre fonction simple est celle qui retourne juste une valeur constante:

La voici:

Listing 1.8 : Code C/C++

```
int f()
{
        return 123;
};
```

Compilons la!

1.4.1 x86

Voici ce que les compilateurs GCC et MSVC produisent sur une plateforme x86:

Listing 1.9 : GCC/MSVC avec optimisation (résultat en sortie de l'assembleur)

```
f:
        mov     eax, 123
        ret
```

Il y a juste deux instructions: la première place la valeur 123 dans le registre EAX, qui est par convention le registre utilisé pour stocker la valeur renvoyée d'une fonction et la seconde est RET, qui retourne l'exécution vers l'appelant.

L'appelant prendra le résultat de cette fonction dans le registre EAX.

1.4.2 ARM

Il y a quelques différences sur la platforme ARM:

Listing 1.10 : avec optimisation Keil 6/2013 (Mode ARM) ASM Output

```
f       PROC
        MOV      r0,#0x7b ; 123
        BX       lr
        ENDP
```

ARM utilise le registre R0 pour renvoyer le résultat d'une fonction, donc 123 est copié dans R0.

Il est à noter que l'instruction MOV est trompeuse pour les plateformes x86 et ARM ISAs.

La donnée n'est en réalité pas *déplacée (moved)* mais *copiée*.

1.4.3 MIPS

La sortie de l'assembleur GCC ci-dessous indique les registres par numéro:

Listing 1.11 : GCC 4.4.5 avec optimisation (résultat en sortie de l'assembleur)

```
        j       $31
        li      $2,123                  # 0x7b
```

…tandis qu'IDA le fait—avec les pseudo noms:

Listing 1.12 : GCC 4.4.5 avec optimisation (IDA)

```
                jr      $ra
                li      $v0, 0x7B
```

Le registre $2 (ou $V0) est utilisé pour stocker la valeur de retour de la fonction. LI signifie “Load Immediate” et est l'équivalent MIPS de MOV.

L'autre instruction est l'instruction de saut (J ou JR) qui retourne le flux d'exécution vers l'appelant.

Vous pouvez vous demander pourquoi la position de l'instruction d'affectation de valeur immédiate (LI) et l'instruction de saut (J ou JR) sont échangées. Ceci est dû à une fonctionnalité du RISC appelée “branch delay slot” (slot de délai de branchement).

La raison de cela est du à une bizarrerie dans l'architecture de certains RISC ISAs et n'est pas importante pour nous. Nous gardons juste en tête qu'en MIPS, l'instruction qui suit une instruction de saut ou de branchement est exécutée *avant* l'instruction de saut ou de branchement elle-même.

Par conséquent, les instructions de branchement échangent toujours leur place avec l'instruction qui doit être exécutée avant.

1.4.4 En pratique

Les fonctions qui retournent simplement 1 (*true*) ou 0 (*false*) sont vraiment fréquentes en pratique.

Les plus petits utilitaires UNIX standard, */bin/true* et */bin/false* renvoient respectivement 0 et 1, comme code de retour. (un code retour de zéro signifie en général succès, non-zéro une erreur).

1.5 Hello, world!

Utilisons le fameux exemple du livre [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)] :

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

1.5.1 x86

MSVC

Compilons-le avec MSVC 2010:

```
cl 1.cpp /Fa1.asm
```

(L'option /Fa indique au compilateur de générer un fichier avec le listing en assembleur)

Listing 1.13 : MSVC 2010

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 0AH, 00H
CONST   ENDS
PUBLIC  _main
EXTRN   _printf:PROC
; Function compile flags: /Odtp
_TEXT   SEGMENT
_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG3830
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

```
_TEXT    ENDS
```

MSVC génère des listings assembleur avec la syntaxe Intel. La différence entre la syntaxe Intel et la syntaxe AT&T sera discutée dans 1.5.1 on page 15.

Le compilateur a généré le fichier object `1.obj`, qui sera lié dans l'exécutable `1.exe`. Dans notre cas, le fichier contient deux segments: CONST (pour les données constantes) et _TEXT (pour le code).

La chaîne `hello, world` en C/C++ a le type `const char[]`[Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)p176, 7.3.2], mais elle n'a pas de nom. Le compilateur doit pouvoir l'utiliser et lui défini donc le nom interne `$SG3830` à cette fin.

C'est pourquoi l'exemple pourrait être récrit comme suit:

```
#include <stdio.h>

const char $SG3830[]="hello, world\n";

int main()
{
    printf($SG3830);
    return 0;
}
```

Retournons au listing assembleur. Comme nous le voyons, la chaîne est terminée avec un octet à zéro, ce qui est le standard pour les chaînes C/C++.

Dans le segment de code, _TEXT, il n'y a qu'une seule fonction: `main()`. La fonction `main()` débute par le code du prologue et se termine par le code de l'épilogue (comme presque toutes les fonctions) [15].

Après le prologue de la fonction nous voyons l'appel à la fonction `printf()` : `CALL _printf`. Avant l'appel, l'adresse de la chaîne (ou un pointeur sur elle) contenant notre message est placée sur la pile avec l'aide de l'instruction PUSH.

Lorsque la fonction `printf()` rend le contrôle à la fonction `main()`, l'adresse de la chaîne (ou un pointeur sur elle) est toujours sur la pile. Comme nous n'en avons plus besoin, le pointeur de pile (pointeur de pile le registre ESP) doit être corrigé.

`ADD ESP, 4` signifie ajouter 4 à la valeur du registre ESP.

Pourquoi 4? puisqu'il s'agit d'un programme 32-bit, nous avons besoin d'exactement 4 octets pour passer une adresse par la pile. S'il s'agissait d'un code x64, nous aurions besoin de 8 octets. `ADD ESP, 4` est effectivement équivalent à `POP register` mais sans utiliser de registre[16].

Pour la même raison, certains compilateurs (comme le compilateur C++ d'Intel) peuvent générer `POP ECX` à la place de `ADD` (e.g., ce comportement peut être observé dans le code d'Oracle RDBMS car il est compilé avec le compilateur C++ d'Intel.

[15] Vous pouvez en lire plus dans la section concernant les prologues et épilogues de fonctions (1.6 on page 41).

[16] Les flags du CPU, toutefois, sont modifiés

Cette instruction a presque le même effet mais le contenu du registre ECX sera écrasé. Le compilateur C++ d'Intel utilise probablement `POP ECX` car l'opcode de cette instruction est plus court que celui de `ADD ESP, x` (1 octet pour POP contre 3 pour ADD).

Voici un exemple d'utilisation de POP à la place de ADD dans Oracle RDBMS :

Listing 1.14 : Oracle RDBMS 10.2 Linux (app.o file)

```
.text:0800029A                 push    ebx
.text:0800029B                 call    qksfroChild
.text:080002A0                 pop     ecx
```

Toutefois, MSVC peut faire de même.

Listing 1.15 : MineSweeper de Windows 7 32-bit

```
.text:0102106F                 push    0
.text:01021071                 call    ds:time
.text:01021077                 pop     ecx
```

Après l'appel de `printf()`, le code C/C++ original contient la déclaration `return 0` —renvoie 0 comme valeur de retour de la fonction `main()`.

Dans le code généré cela est implémenté par l'instruction `XOR EAX, EAX`.

XOR est en fait un simple «OU exclusif (eXclusive OR »[17] mais les compilateurs l'utilisent souvent à la place de `MOV EAX, 0`—à nouveau parce que l'opcode est légèrement plus court (2 octets pour XOR contre 5 pour MOV).

Certains compilateurs génèrent `SUB EAX, EAX`, qui signifie *Soustraire la valeur dans* EAX *de la valeur dans* EAX, ce qui, dans tous les cas, donne zéro.

La dernière instruction RET redonne le contrôle à l'appelant. Habituellement, c'est ce code C/C++ CRT[18], qui, à son tour, redonne le contrôle à l'OS.

GCC

Maintenant compilons le même code C/C++ avec le compilateur GCC 4.4.1 sur Linux: `gcc 1.c -o 1`. Ensuite, avec l'aide du désassembleur IDA, regardons comment la fonction `main()` a été créée. IDA, comme MSVC, utilise la syntaxe Intel[19].

Listing 1.16 : code in IDA

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
```

[17] Wikipédia

[18] C Runtime library

[19] GCC peut aussi produire un listing assembleur utilisant la syntaxe Intel en lui passant les options `-S -masm=intel`.

```
                sub     esp, 10h
                mov     eax, offset aHelloWorld ; "hello, world\n"
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Le résultat est presque le même. L'adresse de la chaîne `hello, world` (stockée dans le segment de donnée) est d'abord chargée dans le registre EAX puis sauvée sur la pile.
En plus, le prologue de la fonction comprend `AND ESP, 0FFFFFFF0h` —cette instruction aligne le registre ESP sur une limite de 16-octet. Ainsi, toutes les valeurs sur la pile seront alignées de la même manière (Le CPU est plus performant si les adresses avec lesquelles il travaille en mémoire sont alignées sur des limites de 4-octet ou 16-octet).

`SUB ESP, 10h` réserve 16 octets sur la pile. Pourtant, comme nous allons le voir, seuls 4 sont nécessaires ici.

C'est parce que la taille de la pile allouée est alignée sur une limite de 16-octet.

L'adresse de la chaîne est (ou un pointeur vers la chaîne) est stockée directement sur la pile sans utiliser l'instruction PUSH. *var_10* —est une variable locale et est aussi un argument pour `printf()`. Lisez à ce propos en dessous.

Ensuite la fonction `printf()` est appelée.

Contrairement à MSVC, lorsque GCC compile sans optimisation, il génère `MOV EAX, 0` au lieu d'un opcode plus court.

La dernière instruction, LEAVE —est équivalente à la paire d'instruction `MOV ESP, EBP` et `POP EBP` —en d'autres mots, cette instruction déplace le pointeur de pile (ESP) et remet le registre EBP dans son état initial. Ceci est nécessaire puisque nous avons modifié les valeurs de ces registres (ESP et EBP) au début de la fonction (en exécutant `MOV EBP, ESP` / `AND ESP, …`).

GCC: Syntaxe AT&T

Voyons comment cela peut-être représenté en langage d'assemblage avec la syntaxe AT&T. Cette syntaxe est bien plus populaire dans le monde UNIX.

Listing 1.17 : compilons avec GCC 4.7.3

```
gcc -S 1_1.c
```

Nous obtenons ceci:

Listing 1.18 : GCC 4.7.3

```
        .file   "1_1.c"
        .section        .rodata
.LC0:
```

```
        .string "hello, world\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section        .note.GNU-stack,"",@progbits
```

Le listing contient beaucoup de macros (qui commencent avec un point). Cela ne nous intéresse pas pour le moment.

Pour l'instant, dans un souci de simplification, nous pouvons les ignorer (excepté la macro *.string* qui encode une séquence de caractères terminée par un octet nul, comme une chaîne C). Ensuite nous verrons cela[20] :

Listing 1.19 : GCC 4.7.3

```
.LC0:
        .string "hello, world\n"
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

Quelques-une des différences majeures entre la syntaxe Intel et AT&T sont:

[20]Cette option de GCC peut être utilisée pour éliminer les macros «non nécessaires» : *-fno-asynchronous-unwind-tables*

- Opérandes source et destination sont écrites dans l'ordre inverse.

  En syntaxe Intel: <instruction> <opérande de destination> <opérande source>.

  En syntaxe AT&T: <instruction> <opérande source> <opérande de destination>.

  Voici un moyen simple de mémoriser la différence: lorsque vous avez affaire avec la syntaxe Intel, vous pouvez imaginer qu'il y a un signe égal (=) entre les opérandes et lorsque vous avez affaire avec la syntaxe AT&T imaginez qu'il y a un flèche droite (→) [21].

- AT&T: Avant les noms de registres, un signe pourcent doit être écrit (%) et avant les nombres, un signe dollar ($). Les parenthèses sont utilisées à la place des crochets.

- AT&T: un suffixe est ajouté à l'instruction pour définir la taille de l'opérande:
  - q — quad (64 bits)
  - l — long (32 bits)
  - w — word (16 bits)
  - b — byte (8 bits)

Retournons au résultat compilé: il est identique à ce que l'on voit dans IDA. Avec une différence subtile: `0FFFFFFF0h` est représenté avec `$-16`. C'est la même chose: `16` dans le système décimal est `0x10` en hexadécimal. `-0x10` est équivalent à `0xFFFFFFF0` (pour un type de donnée sur 32-bit).

Encore une chose: la valeur de retour est mise à 0 en utilisant un `MOV` usuel, pas un `XOR`. `MOV` charge seulement la valeur dans le registre. Le nom est mal choisi (la donnée n'est pas déplacée, mais plutôt copiée). Dans d'autres architectures, cette instruction est nommée «LOAD» ou «STORE» ou quelque chose de similaire.

Modification de chaînes (Win32)

Nous pouvons facilement trouver la chaîne "hello, world" dans l'exécutable en utilisant Hiew:

[21]À propos, dans certaine fonction C standard (e.g., memcpy(), strcpy()) les arguments sont listés de la même manière que dans la syntaxe Intel: en premier se trouve le pointeur du bloc mémoire de destination, et ensuite le pointeur sur le bloc mémoire source.

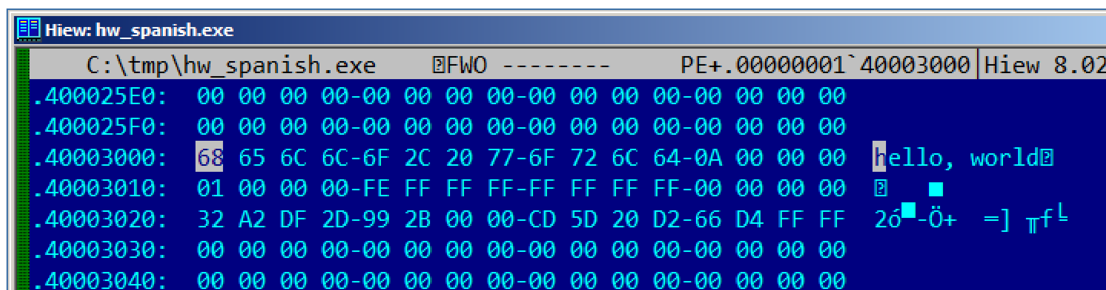


Fig. 1.1: Hiew

Et nous pouvons essayer de traduire notre message en espagnol:

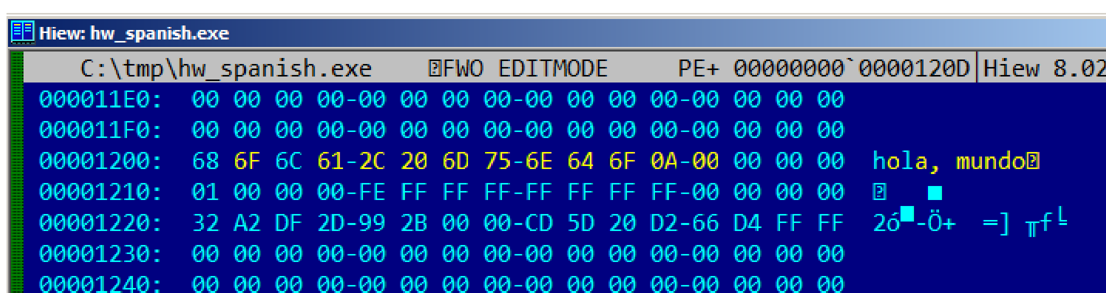


Fig. 1.2: Hiew

Le texte en espagnol est un octet plus court que celui en anglais, nous ajoutons l'octet 0x0A à la fin (\n) ainsi qu'un octet à zéro.

Ça fonctionne.

Comment faire si nous voulons insérer un message plus long ? Il y a quelques octets à zéro après le texte original en anglais. Il est difficile de dire s'ils peuvent être écrasés: ils peuvent être utilisés quelque part dans du code CRT, ou pas. De toutes façons, écrasez-les seulement si vous savez vraiment ce que vous faîtes.

Modification de chaînes (Linux x64)

Essayons de modifier un exécutable Linux x64 en utilisant rada.re :

Listing 1.20 : rada.re session

```
dennis@bigbox ~/tmp % gcc hw.c

dennis@bigbox ~/tmp % radare2 a.out
 -- SHALL WE PLAY A GAME?
[0x00400430]> / hello
Searching 5 bytes from 0x00400000 to 0x00601040: 68 65 6c 6c 6f
```

```
Searching 5 bytes in [0x400000-0x601040]
hits: 1
0x004005c4 hit0_0 .HHhello, world;0.

[0x00400430]> s 0x004005c4

[0x004005c4]> px
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x004005c4  6865 6c6c 6f2c 2077 6f72 6c64 0000 0000  hello, world....
0x004005d4  011b 033b 3000 0000 0500 0000 1cfe ffff  ...;0...........
0x004005e4  7c00 0000 5cfe ffff 4c00 0000 52ff ffff  |...\...L...R...
0x004005f4  a400 0000 6cff ffff c400 0000 dcff ffff  ....l...........
0x00400604  0c01 0000 1400 0000 0000 0000 017a 5200  .............zR.
0x00400614  0178 1001 1b0c 0708 9001 0710 1400 0000  .x..............
0x00400624  1c00 0000 08fe ffff 2a00 0000 0000 0000  ........*.......
0x00400634  0000 0000 1400 0000 0000 0000 017a 5200  .............zR.
0x00400644  0178 1001 1b0c 0708 9001 0000 2400 0000  .x..........$...
0x00400654  1c00 0000 98fd ffff 3000 0000 000e 1046  ........0......F
0x00400664  0e18 4a0f 0b77 0880 003f 1a3b 2a33 2422  ..J..w...?.;*3$"
0x00400674  0000 0000 1c00 0000 4400 0000 a6fe ffff  ........D.......
0x00400684  1500 0000 0041 0e10 8602 430d 0650 0c07  .....A....C..P..
0x00400694  0800 0000 4400 0000 6400 0000 a0fe ffff  ....D...d.......
0x004006a4  6500 0000 0042 0e10 8f02 420e 188e 0345  e....B....B....E
0x004006b4  0e20 8d04 420e 288c 0548 0e30 8606 480e  . ..B.(..H.0..H.

[0x004005c4]> oo+
File a.out reopened in read-write mode

[0x004005c4]> w hola, mundo\x00

[0x004005c4]> q

dennis@bigbox ~/tmp % ./a.out
hola, mundo
```

Ce que je fais ici: je cherche la chaîne «hello» en utilisant la commande `/`, ensuite je déplace le *curseur* (ou *seek* selon la terminologie de rada.re) à cette adresse. Je veux être certain d'être à la bonne adresse: px affiche les octets ici. oo+ passe rada.re en mode *read-write*. w écrit une chaîne ASCII à la *seek* (*position*) courante. Notez le `\00` à la fin–c'est l'octet à zéro. q quitte.

subsubsectionCeci est une histoire vraie de modification de logiciel

Un logiciel de traitement d'image, lorsqu'il n'était pas enregistré, ajoutait un tatouage numérique comme "Cette image a été traitée par la version d'évaluation de [nom du logiciel]", à travers l'image. Nous avons essayé au hasard: nous avons trouvé cette chaîne dans le fichier exécutable et avons mis des espaces à la place. Le tatouage a disparu. Techniquement parlant, il continuait d'apparaître. Avec l'aide des fonctions Qt, le tatouage numérique était toujours ajouté à l'image résultante. Mais ajouter des espaces n'altérait pas l'image elle-même...

Traduction de logiciel à l'ère MS-DOS

La méthode que je viens de décrire était couramment employée pour traduire des logiciels sous MS-DOS en russe dans les années 1980 et 1990. Cette technique est accessible même pour ceux qui ne connaissent pas le code machine et les formats de fichier exécutable. La nouvelle chaîne ne doit être pas être plus longue que l'ancienne, car il y a un risque d'écraser une autre valeur ou du code ici. Les mots et les phrases russes sont en général un peu plus longs qu'en anglais, c'est pourquoi les logiciels *traduits* sont pleins d'acronymes sibyllins et d'abréviations difficilement lisibles.

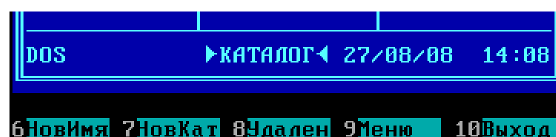


Fig. 1.3: Norton Commander 5.51 *localisé*

Peut-être que cela s'est produit pour d'autres langages durant cette période.

En ce qui concerne Delphi, la taille de la chaîne de caractères doit elle aussi être ajustée.

1.5.2 x86-64

MSVC: x86-64

Essayons MSVC 64-bit:

Listing 1.21 : MSVC 2012 x64

```
$SG2989 DB      'hello, world', 0AH, 00H

main    PROC
        sub     rsp, 40
        lea     rcx, OFFSET FLAT:$SG2989
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP
```

En x86-64, tous les registres ont été étendus à 64-bit et leurs noms ont maintenant le préfixe R-. Afin d'utiliser la pile moins souvent (en d'autres termes, pour accéder moins souvent à la mémoire externe/au cache), il existe un moyen répandu de passer les arguments aux fonctions par les registres (*fastcall*) 6.1.3 on page 955. I.e., une partie des arguments de la fonction est passée par les registres, le reste—par la pile. En Win64, 4 arguments de fonction sont passés dans les registres RCX, RDX, R8, R9. C'est ce que l'on voit ci-dessus: un pointeur sur la chaîne pour `printf()` est passé non pas par la pile, mais par le registre RCX. Les pointeurs font maintenant 64-bit, ils sont donc passés dans les registres 64-bit (qui ont le préfixe R-). Toutefois, pour la

rétrocompatibilité, il est toujours possible d'accéder à la partie 32-bits des registres, en utilisant le préfixe E-. Voici à quoi ressemblent les registres RAX/EAX/AX/AL en x86-64:

Octet d'indice							
7	6	5	4	3	2	1	0
RAX^{x64}							
				EAX			
						AX	
						AH	AL

La fonction `main()` renvoie un type *int*, qui est, en C/C++, pour une meilleure rétrocompatibilité et portabilité, toujours 32-bit, c'est pourquoi le registre EAX est mis à zéro à la fin de la fonction (i.e., la partie 32-bit du registre) au lieu de RAX. Il y aussi 40 octets alloués sur la pile locale. Cela est appelé le «shadow space», dont nous parlerons plus tard: 1.14.2 on page 137.

GCC: x86-64

Essayons GCC sur un Linux 64-bit:

Listing 1.22 : GCC 4.4.6 x64

```
.string "hello, world\n"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0 ; "hello, world\n"
        xor     eax, eax  ; nombre de registres vectoriels
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Une méthode de passage des arguments à la fonction dans des registres est aussi utilisée sur Linux, *BSD et Mac OS X est [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [22]. Linux, *BSD et Mac OS X utilisent aussi une méthode pour passer les arguments d'une fonction par les registres: [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [23].

Les 6 premiers arguments sont passés dans les registres RDI, RSI, RDX, RCX, R8, R9 et les autres—par la pile.

Donc le pointeur sur la chaîne est passé dans EDI (la partie 32-bit du registre). Mais pourquoi ne pas utiliser la partie 64-bit, RDI ?

Il est important de garder à l'esprit que toutes les instructions MOV en mode 64-bit qui écrivent quelque chose dans la partie 32-bit inférieuaer du registre efface également

[22]Aussi disponible en https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

[23]Aussi disponible en https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

les 32-bit supérieurs (comme indiqué dans les manuels Intel: 12.1.4 on page 1315). I.e., l'instruction MOV EAX, 011223344h écrit correctement une valeur dans RAX, puisque que les bits supérieurs sont mis à zéro.

Si nous ouvrons le fichier objet compilé (.o), nous pouvons voir tous les opcodes des instructions [24] :

Listing 1.23 : GCC 4.4.6 x64

```
.text:00000000004004D0                    main  proc near
.text:00000000004004D0 48 83 EC 08        sub     rsp, 8
.text:00000000004004D4 BF E8 05 40 00     mov     edi, offset format ; "hello,
    world\n"
.text:00000000004004D9 31 C0              xor     eax, eax
.text:00000000004004DB E8 D8 FE FF FF     call    _printf
.text:00000000004004E0 31 C0              xor     eax, eax
.text:00000000004004E2 48 83 C4 08        add     rsp, 8
.text:00000000004004E6 C3                 retn
.text:00000000004004E6                    main  endp
```

Comme on le voit, l'instruction qui écrit dans EDI en 0x4004D4 occupe 5 octets. La même instruction qui écrit une valeur sur 64-bit dans RDI occupe 7 octets. Il semble que GCC essaye d'économiser un peu d'espace. En outre, cela permet d'être sûr que le segment de données contenant la chaîne ne sera pas alloué à une adresse supérieure à 4 GiB.

Nous voyons aussi que le registre EAX est mis à zéro avant l'appel à la fonction printf(). Ceci, car conformément à l'ABI[25] standard mentionnée plus haut, le nombre de registres vectoriel utilisés est passé dans EAX sur les systèmes *NIX en x86-64.

Modification d'adresse (Win64)

Lorsque notre exemple est compilé sous MSVC 2013 avec l'option /MD (générant un exécutable plus petit du fait du lien avec MSVCR*.DLL), la fonction main() vient en premier et est trouvée facilement:

[24] Ceci doit être activé dans **Options** → **Disassembly** → **Number of opcode bytes**

[25] Application Binary Interface

```
Hiew: hw2.exe
    C:\tmp\hw2.exe              FWO EDITMODE a64 PE+ 00000000`00000404 Hiew 8.02 (c)SEN
 00000400: 4883EC28                          sub         rsp,028 ;'('
 00000404: 488D0DF51F0000                    lea         rcx,[000002400]
 0000040B: FF15D7100000                      call        q,[0000014E8]
 00000411: 33C0                              xor         eax,eax
 00000413: 4883C428                          add         rsp,028 ;'('
 00000417: C3                                retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
 00000418: 4883EC28                          sub         rsp,028 ;'('
 0000041C: B84D5A0000                        mov         eax,000005A4D ;'  ZM'
 00000421: 663905D8EFFFFF                    cmp         [-000000C00],ax
 00000428: 7404                              jz          00000042E
 000004
 000004     lea         rcx,[0000000000002401]
 000004
 000004               CommandSelect: Off
 000004
 0000043F: 813850450000                      cmp         d,[rax],000004550 ;'  EP'
 00000445: 75E3                              jnz         00000042A
 00000447: B90B020000                        mov         ecx,00000020B
 0000044C: 66394818                          cmp         [rax][018],cx
 00000450: 75D8                              jnz         00000042A
 00000452: 33C9                              xor         ecx,ecx
 00000454: 83B8840000000E                    cmp         d,[rax][000000084],00E
 0000045B: 7609                              jbe         000000466
 0000045D: 3988F8000000                      cmp         [rax][0000000F8],ecx
 1Help   2       3       4Select 5       6       7       8       9       10      11
```

Fig. 1.4: Hiew

A titre expérimental, nous pouvons incrémenter l'adresse du pointeur de 1:

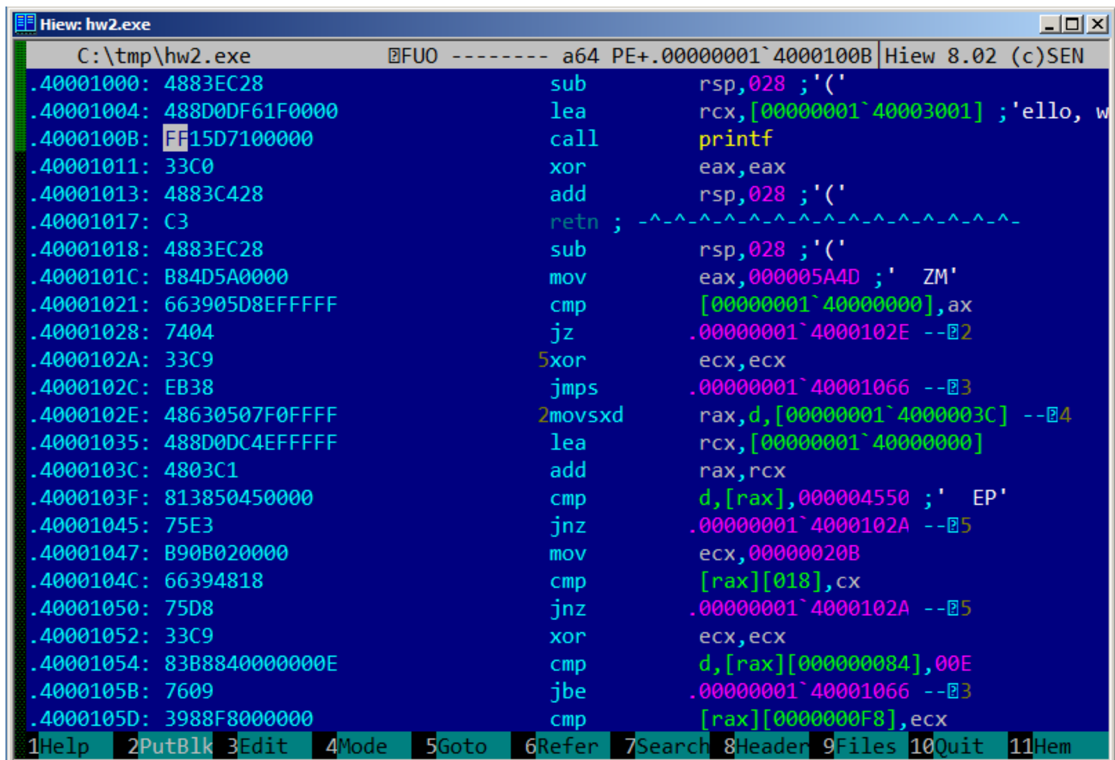


Fig. 1.5: Hiew

Hiew montre la chaîne «ello, world». Et lorsque nous lançons l'exécutable modifié, la chaîne raccourcie est affichée.

Utiliser une autre chaîne d'un binaire (Linux x64)

Le fichier binaire que j'obtiens en compilant notre exemple avec GCC 5.4.0 sur un système Linux x64 contient de nombreuses autres chaînes: la plupart sont des noms de fonction et de bibliothèque importées.

Je lance *objdump* pour voir le contenu de toutes les sections du fichier compilé:

```
$ objdump -s a.out

a.out:     file format elf64-x86-64

Contents of section .interp:
 400238 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-
 400248 7838362d 36342e73 6f2e3200           x86-64.so.2.
Contents of section .note.ABI-tag:
 400254 04000000 10000000 01000000 474e5500  ............GNU.
 400264 00000000 02000000 06000000 20000000  ............ ...
Contents of section .note.gnu.build-id:
```

```
 400274 04000000 14000000 03000000 474e5500  ............GNU.
 400284 fe461178 5bb710b4 bbf2aca8 5ec1ec10  .F.x[.......^...
 400294 cf3f7ae4                             .?z.

...
```

Ce n'est pas un problème de passer l'adresse de la chaîne «/lib64/ld-linux-x86-64.so.2 » à l'appel de `printf()` :

```
#include <stdio.h>

int main()
{
    printf(0x400238);
    return 0;
}
```

Difficile à croire, ce code affiche la chaîne mentionnée.

Changez l'adresse en `0x400260`, et la chaîne «GNU » sera affichée. L'adresse est valable pour cette version spécifique de GCC, outils GNU, etc. Sur votre système, l'exécutable peut être légèrement différent, et toutes les adresses seront différentes. Ainsi, ajouter/supprimer du code à/de ce code source va probablement décaler les adresses en arrière et avant.

1.5.3 ARM

Pour mes expérimentations avec les processeurs ARM, différents compilateurs ont été utilisés:

- Très courant dans le monde de l'embarqué: Keil Release 6/2013.
- Apple Xcode 4.6.3 IDE avec le compilateur LLVM-GCC 4.2 [26]
- GCC 4.9 (Linaro) (pour ARM64), disponible comme exécutable win32 ici `http://www.linaro.org/projects/armv8/`.

C'est du code ARM 32-bit qui est utilisé (également pour les modes Thumb et Thumb-2) dans tous les cas dans ce livre, sauf mention contraire.

sans optimisation Keil 6/2013 (Mode ARM)

Commençons par compiler notre exemple avec Keil:

```
armcc.exe --arm --c90 -O0 1.c
```

Le compilateur *armcc* produit un listing assembleur en syntaxe Intel, mais il dispose de macros de haut niveau liées au processeur ARM[27]. Comme il est plus important

[26] C'est ainsi: Apple Xcode 4.6.3 utilise les composants open-source GCC comme front-end et LLVM comme générateur de code

[27] e.g. les instructions PUSH/POP manquent en mode ARM

pour nous de voir les instructions «telles quelles», nous regardons le résultat compilé dans IDA.

Listing 1.24 : sans optimisation Keil 6/2013 (Mode ARM) IDA

```
.text:00000000                 main
.text:00000000 10 40 2D E9     STMFD   SP!, {R4,LR}
.text:00000004 1E 0E 8F E2     ADR     R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB     BL      __2printf
.text:0000000C 00 00 A0 E3     MOV     R0, #0
.text:00000010 10 80 BD E8     LDMFD   SP!, {R4,PC}

.text:000001EC 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF:
    main+4
```

Dans l'exemple, nous voyons facilement que chaque instruction a une taille de 4 octets. En effet, nous avons compilé notre code en mode ARM, pas pour Thumb.

La toute première instruction, `STMFD SP!, {R4,LR}`[28], fonctionne comme une instruction PUSH en x86, écrivant la valeur de deux registres (R4 et LR) sur la pile.

En effet, dans le listing de la sortie du compilateur *armcc*, dans un souci de simplification, il montre l'instruction `PUSH {r4,lr}`. Mais ce n'est pas très précis. L'instruction PUSH est seulement disponible dans le mode Thumb. Donc, pour rendre les choses moins confuses, nous faisons cela dans IDA.

Cette instruction décrémente d'abord le pointeur de pile SP[30] pour qu'il pointe sur de l'espace libre pour de nouvelles entrées, ensuite elle sauve les valeurs des registres R4 et LR à cette adresse.

Cette instruction (comme l'instruction PUSH en mode Thumb) est capable de sauvegarder plusieurs valeurs de registre à la fois, ce qui peut être très utile. À propos, elle n'a pas d'équivalent en x86. On peut noter que l'instruction STMFD est une généralisation de l'instruction PUSH (étendant ses fonctionnalités), puisqu'elle peut travailler avec n'importe quel registre, pas seulement avec SP. En d'autres mots, l'instruction STMFD peut être utilisée pour stocker un ensemble de registres à une adresse donnée.

L'instruction `ADR R0, aHelloWorld` ajoute ou soustrait la valeur dans le registre PC[31] à l'offset où la chaîne `hello, world` se trouve. On peut se demander comment le registre PC est utilisé ici ? C'est appelé du «code indépendant de la position»[32].

Un tel code peut être exécuté à n'importe quelle adresse en mémoire. En d'autres mots, c'est un adressage PC-relatif. L'instruction ADR prend en compte la différence entre l'adresse de cette instruction et l'adresse où est située la chaîne. Cette différence (offset) est toujours la même, peu importe à quelle adresse notre code est chargé par l'OS. C'est pourquoi tout ce dont nous avons besoin est d'ajouter l'adresse de l'instruction courante (du PC) pour obtenir l'adresse absolue en mémoire de notre chaîne C.

[28] STMFD[29]
[30] pointeur de pile. SP/ESP/RSP dans x86/x64. SP dans ARM.
[31] Program Counter. IP/EIP/RIP dans x86/64. PC dans ARM.
[32] Lire à ce propos la section(6.4.1 on page 974)

L'instruction BL __2printf[33] appelle la fonction printf(). Voici comment fonctionne cette instruction:

- sauve l'adresse suivant l'instruction BL (0xC) dans LR ;
- puis passe le contrôle à printf() en écrivant son adresse dans le registre PC.

Lorsque la fonction printf() termine son exécution elle doit avoir savoir où elle doit redonner le contrôle. C'est pourquoi chaque fonction passe le contrôle à l'adresse se trouvant dans le registre LR.

C'est une différence entre un processeur RISC « pur » comme ARM et un processeur CISC[34] comme x86, où l'adresse de retour est en général sauvée sur la pile. Pour aller plus loin, lire la section (1.9 on page 42) suivante.

À propos, une adresse absolue ou un offset de 32-bit ne peuvent être encodés dans l'instruction 32-bit BL car il n'y a qu'un espace de 24 bits. Comme nous devons nous en souvenir, toutes les instructions ont une taille de 4 octets (32 bits). Par conséquent, elles ne peuvent se trouver qu'à des adresses alignées dur des limites de 4 octets. Cela implique que les 2 derniers bits de l'adresse d'une instruction (qui sont toujours des bits à zéro) peuvent être omis. En résumé, nous avons 26 bits pour encoder l'offset. C'est assez pour encoder $current_PC \pm \approx 32M$.

Ensuite, l'instruction MOV R0, #0[35] écrit juste 0 dans le registre R0. C'est parce que notre fonction C renvoie 0 et la valeur de retour doit être mise dans le registre R0.

La dernière instruction est LDMFD SP!, R4,PC[36]. Elle prend des valeurs sur la pile (ou de toute autre endroit en mémoire) afin de les sauver dans R4 et PC, et incrémente le pointeur de pile SP. Cela fonctionne ici comme POP.
N.B. La toute première instruction STMFD a sauvé la paire de registres R4 et LR sur la pile, mais R4 et PC sont *restaurés* pendant l'exécution de LDMFD.

Comme nous le savons déjà, l'adresse où chaque fonction doit redonner le contrôle est usuellement sauvée dans le registre LR. La toute première instruction sauve sa valeur sur la pile car le même registre va être utilisé par notre fonction main() lors de l'appel à printf(). A la fin de la fonction, cette valeur peut être écrite directement dans le registre PC, passant ainsi le contrôle là où notre fonction a été appelée. Comme main() est en général la première fonction en C/C++, le contrôle sera redonné au chargeur de l'OS ou a un point dans un CRT, ou quelque chose comme ça.

Tout cela permet d'omettre l'instruction BX LR à la fin de la fonction.

DCB est une directive du langage d'assemblage définissant un tableau d'octets ou des chaînes ASCII, proche de la directive DB dans le langage d'assemblage x86.

sans optimisation Keil 6/2013 (Mode Thumb)

Compilons le même exemple en utilisant keil en mode Thumb:

33Branch with Link
34Complex Instruction Set Computing
35Signifiant MOVe
36LDMFD[37] est l'instruction inverse de STMFD

```
armcc.exe --thumb --c90 -O0 1.c
```

Nous obtenons (dans IDA) :

Listing 1.25 : sans optimisation Keil 6/2013 (Mode Thumb) + IDA

```
.text:00000000             main
.text:00000000 10 B5          PUSH    {R4,LR}
.text:00000002 C0 A0          ADR     R0, aHelloWorld ; "hello, world"
.text:00000004 06 F0 2E F9    BL      __2printf
.text:00000008 00 20          MOVS    R0, #0
.text:0000000A 10 BD          POP     {R4,PC}

.text:00000304 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF:
    main+2
```

Nous pouvons repérer facilement les opcodes sur 2 octets (16-bit). C'est, comme déjà noté, Thumb. L'instruction BL, toutefois, consiste en deux instructions 16-bit. C'est parce qu'il est impossible de charger un offset pour la fonction `printf()` en utilisant seulement le petit espace dans un opcode 16-bit. Donc, la première instruction 16-bit charge les 10 bits supérieurs de l'offset et la seconde instruction les 11 bits inférieurs de l'offset.

Comme il a été écrit, toutes les instructions en mode Thumb ont une taille de 2 octets (ou 16 bits). Cela implique qu'il impossible pour une instruction Thumb d'être à une adresse impaire, quelle qu'elle soit. En tenant compte de cela, le dernier bit de l'adresse peut être omis lors de l'encodage des instructions.

En résumé, l'instruction Thumb BL peut encoder une adresse en $current_PC \pm \approx 2M$.

Comme pour les autres instructions dans la fonction: PUSH et POP fonctionnent ici comme les instructions décrites STMFD/LDMFD seul le registre SP n'est pas mentionné explicitement ici. ADR fonctionne comme dans l'exemple précédent. MOVS écrit 0 dans le registre R0 afin de renvoyer zéro.

avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Xcode 4.6.3 sans l'option d'optimisation produit beaucoup de code redondant c'est pourquoi nous allons étudier le code généré avec optimisation, où le nombre d'instruction est aussi petit que possible, en mettant l'option -O3 du compilateur.

Listing 1.26 : avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
__text:000028C4             _hello_world
__text:000028C4 80 40 2D E9   STMFD          SP!, {R7,LR}
__text:000028C8 86 06 01 E3   MOV            R0, #0x1686
__text:000028CC 0D 70 A0 E1   MOV            R7, SP
__text:000028D0 00 00 40 E3   MOVT           R0, #0
__text:000028D4 00 00 8F E0   ADD            R0, PC, R0
__text:000028D8 C3 05 00 EB   BL             _puts
__text:000028DC 00 00 A0 E3   MOV            R0, #0
__text:000028E0 80 80 BD E8   LDMFD          SP!, {R7,PC}

__cstring:00003F62 48 65 6C 6C+aHelloWorld_0  DCB "Hello world!",0
```

Les instructions STMFD et LDMFD nous sont déjà familières.

L'instruction MOV écrit simplement le nombre 0x1686 dans le registre R0. C'est l'offset pointant sur la chaîne «Hello world! ».

Le registre R7 (tel qu'il est standardisé dans [*iOS ABI Function Call Guide*, (2010)][38]) est un pointeur de frame. Voir plus loin.

L'instruction MOVT R0, #0 (MOVe Top) écrit 0 dans les 16 bits de poids fort du registre. Le problème ici est que l'instruction générique MOV en mode ARM peut n'écrire que dans les 16 bits de poids faible du registre.

Il faut garder à l'esprit que tout les opcodes d'instruction en mode ARM sont limités en taille à 32 bits. Bien sûr, cette limitation n'est pas relative au déplacement de données entre registres. C'est pourquoi une instruction supplémentaire existe MOVT pour écrire dans les bits de la partie haute (de 16 à 31 inclus). Son usage ici, toutefois, est redondant car l'instruction MOV R0, #0x1686 ci dessus a effacé la partie haute du registre. C'est soi-disant un défaut du compilateur.

L'instruction ADD R0, PC, R0 ajoute la valeur dans PC à celle de R0, pour calculer l'adresse absolue de la chaîne «Hello world! ». Comme nous l'avons déjà vu, il s'agit de «code indépendant de la position » donc la correction est essentielle ici.

L'instruction BL appelle la fonction puts() au lieu de printf().

LLVM a remplacé le premier appel à printf() par un à puts(). Effectivement: printf() avec un unique argument est presque analogue à puts().

Presque, car les deux fonctions produisent le même résultat uniquement dans le cas où la chaîne ne contient pas d'identifiants de format débutant par %. Dans le cas où elle en contient, l'effet de ces deux fonctions est différent[39].

Pourquoi est-ce que le compilateur a remplacé printf() par puts() ? Probablement car puts() est plus rapide[40].

Car il envoie seulement les caractères dans sortie standard sans comparer chacun d'entre eux avec le symbole %.

Ensuite, nous voyons l'instruction familière MOV R0, #0 pour mettre le registre R0 à 0.

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

Par défaut Xcode 4.6.3 génère du code pour Thumb-2 de cette manière:

Listing 1.27 : avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
__text:00002B6C                    _hello_world
__text:00002B6C 80 B5           PUSH            {R7,LR}
__text:00002B6E 41 F2 D8 30     MOVW            R0, #0x13D8
__text:00002B72 6F 46           MOV             R7, SP
```

[38]Aussi disponible en http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf

[39]Il est à noter que puts() ne nécessite pas un '\n' symbole de retour à la ligne à la fin de la chaîne, donc nous ne le voyons pas ici.

[40]ciselant.de/projects/gcc_printf/gcc_printf.html

```
__text:00002B74 C0 F2 00 00    MOVT.W         R0, #0
__text:00002B78 78 44          ADD            R0, PC
__text:00002B7A 01 F0 38 EA    BLX            _puts
__text:00002B7E 00 20          MOVS           R0, #0
__text:00002B80 80 BD          POP            {R7,PC}

...

__cstring:00003E70 48 65 6C 6C 6F 20+aHelloWorld  DCB "Hello world!",0xA,0
```

Les instructions BL et BLX en mode Thumb, comme on s'en souvient, sont encodées comme une paire d'instructions 16 bits. En Thumb-2 ces opcodes *substituts* sont étendus de telle sorte que les nouvelles instructions puissent être encodées comme des instructions 32-bit.

C'est évident en considérant que les opcodes des instructions Thumb-2 commencent toujours avec `0xFx` ou `0xEx`.

Mais dans le listing d'IDA les octets d'opcodes sont échangés car pour le processeur ARM les instructions sont encodées comme ceci: dernier octet en premier et ensuite le premier (pour les modes Thumb et Thumb-2) ou pour les instructions en mode ARM le quatrième octet vient en premier, ensuite le troisième, puis le second et enfin le premier (à cause des différents endianness).

C'est ainsi que les octets se trouvent dans le listing d'IDA:

- pour les modes ARM et ARM64: 4-3-2-1;
- pour le mode Thumb: 2-1;
- pour les paires d'instructions 16-bit en mode Thumb-2: 2-1-4-3.

Donc, comme on peut le voir, les instructions `MOVW`, `MOVT.W` et `BLX` commencent par `0xFx`.

Une des instructions Thumb-2 est `MOVW R0, #0x13D8` —elle stocke une valeur 16-bit dans la partie inférieure du registre `R0`, effaçant les bits supérieurs.

Aussi, `MOVT.W R0, #0` fonctionne comme `MOVT` de l'exemple précédent mais il fonctionne en Thumb-2.

Parmi les autres différences, l'instruction `BLX` est utilisée dans ce cas à à la place de `BL`.

La différence est que, en plus de sauver RA[41] dans le registre LR et de passer le contrôle à la fonction `puts()`, le processeur change du mode Thumb/Thumb-2 au mode ARM (ou inversement).

Cette instruction est placée ici, car l'instruction à laquelle est passée le contrôle ressemble à (c'est encodé en mode ARM) :

```
__symbolstub1:00003FEC _puts           ; CODE XREF: _hello_world+E
__symbolstub1:00003FEC 44 F0 9F E5     LDR  PC, =__imp__puts
```

[41] Adresse de retour

Il s'agit principalement d'un saut à l'endroit où l'adresse de `puts()` est écrit dans la section import.

Mais alors, le lecteur attentif pourrait demander: pourquoi ne pas appeler `puts()` depuis l'endroit dans le code où on en a besoin ?

Parce que ce n'est pas très efficace en terme d'espace.

Presque tous les programmes utilisent des bibliothèques dynamiques externes (comme les DLL sous Windows, les .so sous *NIX ou les .dylib sous Mac OS X). Les bibliothèques dynamiques contiennent les bibliothèques fréquemment utilisées, incluant la fonction C standard `puts()`.

Dans un fichier binaire exécutable (Windows PE .exe, ELF ou Mach-O) se trouve une section d'import. Il s'agit d'une liste des symboles (fonctions ou variables globales) importées depuis des modules externes avec le nom des modules eux-même.

Le chargeur de l'OS charge tous les modules dont il a besoin, tout en énumérant les symboles d'import dans le module primaire, il détermine l'adresse correcte de chaque symbole.

Dans notre cas, *__imp__puts* est une variable 32-bit utilisée par le chargeur de l'OS pour sauver l'adresse correcte d'une fonction dans une bibliothèque externe. Ensuite l'instruction LDR lit la valeur 32-bit depuis cette variable et l'écrit dans le registre PC, lui passant le contrôle.

Donc, pour réduire le temps dont le chargeur de l'OS à besoin pour réaliser cette procédure, c'est une bonne idée d'écrire l'adresse de chaque symbole une seule fois, à une place dédiée.

À côté de ça, comme nous l'avons déjà compris, il est impossible de charger une valeur 32-bit dans un registre en utilisant seulement une instruction sans un accès mémoire.

Donc, la solution optimale est d'allouer une fonction séparée fonctionnant en mode ARM avec le seul but de passer le contrôle à la bibliothèque dynamique et ensuite de sauter à cette petite fonction d'une instruction (ainsi appelée fonction thunk) depuis le code Thumb.

À propos, dans l'exemple précédent (compilé en mode ARM), le contrôle est passé par BL à la même fonction thunk. Le mode du processeur, toutefois, n'est pas échangé (d'où l'absence d'un « X » dans le mnémonique de l'instruction).

Plus à propos des fonctions thunk

Les fonctions thunk sont difficile à comprendre, apparemment, à cause d'un mauvais nom. La manière la plus simple est de les voir comme des adaptateurs ou des convertisseurs d'un type jack à un autre. Par exemple, un adaptateur permettant l'insertion d'un cordon électrique britannique sur une prise murale américaine, ou vice-versa. Les fonctions thunk sont parfois appelées *wrappers*.

Voici quelques autres descriptions de ces fonctions:

> “Un morceau de code qui fournit une adresse:”, d’après P. Z. Ingerman, qui inventa thunk en 1961 comme un moyen de lier les paramètres réels à leur définition formelle dans les appels de procédures en Algol-60. Si une procédure est appelée avec une expression à la place d’un paramètre formel, le compilateur génère un thunk qui calcule l’expression et laisse l’adresse du résultat dans une place standard.
>
> …
>
> Microsoft et IBM ont tous les deux défini, dans systèmes basés sur Intel, un ”environnement 16-bit” (avec leurs horribles registres de segment et la limite des adresses à 64K) et un ”environnement 32-bit” (avec un adressage linéaire et une gestion semi-réelle de la mémoire). Les deux environnements peuvent fonctionner sur le même ordinateur et OS (grâce à ce qui est appelé, dans le monde Microsoft, WOW qui signifie Windows dans Windows). MS et IBM ont tous deux décidé que le procédé de passer de 16-bit à 32-bit et vice-versa est appelé un ”thunk”; pour Window 95, il y a même un outil, THUNK.EXE, appelé un ”compilateur thunk”.

(The Jargon File)

Nous pouvons trouver un autre exemple dans la bibliothèque LAPCAK—un “Linear Algebra PACKage” écrit en FORTRAN. Les développeurs C/C++ veulent aussi utiliser LAPACK, mais c’est un non-sens de la récrire en C/C++ et de maintenir plusieurs versions. Donc, il y a des petites fonctions que l’on peut invoquer depuis un environnement C/C++, qui font, à leur tour, des appels aux fonctions FORTRAN, et qui font presque tout le reste:

```
double Blas_Dot_Prod(const LaVectorDouble &dx, const LaVectorDouble &dy)
{
    assert(dx.size()==dy.size());
    integer n = dx.size();
    integer incx = dx.inc(), incy = dy.inc();

    return F77NAME(ddot)(&n, &dx(0), &incx, &dy(0), &incy);
}
```

Donc, ce genre de fonctions est appelé “wrappers”.

ARM64

GCC

Compilons l’exemple en utilisant GCC 4.8.1 en ARM64:

Listing 1.28 : GCC 4.8.1 sans optimisation + objdump

```
0000000000400590 <main>:
  400590:       a9bf7bfd        stp     x29, x30, [sp,#-16]!
  400594:       910003fd        mov     x29, sp
```

```
  400598:       90000000        adrp    x0, 400000 <_init-0x3b8>
  40059c:       91192000        add     x0, x0, #0x648
  4005a0:       97ffffa0        bl      400420 <puts@plt>
  4005a4:       52800000        mov     w0, #0x0  // #0
  4005a8:       a8c17bfd        ldp     x29, x30, [sp],#16
  4005ac:       d65f03c0        ret

...

Contents of section .rodata:
 400640 01000200 00000000 48656c6c 6f210a00  ........Hello!..
```

Il n'y a pas de mode Thumb ou Thumb-2 en ARM64, seulement en ARM, donc il n'y a que des instructions 32-bit. Le nombre de registres a doublé: .2.4 on page 1351. Les registres 64-bit ont le préfixe `X-`, tandis que leurs partie 32-bit basse—`W-`.

L'instruction STP (*Store Pair* stocke une paire) sauve deux registres sur la pile simultanément: X29 et X30.

Bien sûr, cette instruction peut sauvegarder cette paire à n'importe quelle endroit en mémoire, mais le registre SP est spécifié ici, donc la paire est sauvé sur le pile.

Les registres ARM64 font 64-bit, chacun a une taille de 8 octets, donc il faut 16 octets pour sauver deux registres.

Le point d'exclamation ("!") après l'opérande signifie que 16 octets doivent d'abord être soustrait de SP, et ensuite les valeurs de la paire de registres peuvent être écrites sur la pile. Ceci est appelé le *pre-index*. À propos de la différence entre *post-index* et *pre-index* lisez ceci: 1.39.2 on page 568.

Dans la gamme plus connue du x86, la première instruction est analogue à la paire `PUSH X29` et `PUSH X30`. En ARM64, X29 est utilisé comme FP[42] et X30 comme LR, c'est pourquoi ils sont sauvegardés dans le prologue de la fonction et remis dans l'épilogue.

La seconde instruction copie SP dans X29 (ou FP). Cela sert à préparer la pile de la fonction.

Les instructions `ADRP` et `ADD` sont utilisées pour remplir l'adresse de la chaîne «Hello! » dans le registre `X0`, car le premier argument de la fonction est passé dans ce registre. Il n'y a pas d'instruction, quelqu'elle soit, en ARM qui puisse stocker un nombre large dans un registre (car la longueur des instructions est limitée à 4 octets, cf: 1.39.3 on page 570). Plusieurs instructions doivent donc être utilisées. La première instruction (`ADRP`) écrit l'adresse de la page de 4KiB, où se trouve la chaîne, dans `X0`, et la seconde (`ADD`) ajoute simplement le reste de l'adresse. Plus d'information ici: 1.39.4 on page 572.

`0x400000 + 0x648 = 0x400648`, et nous voyons notre chaîne C «Hello! » dans le `.rodata` segment des données à cette adresse.

`puts()` est appelée après en utilisant l'instruction `BL`. Cela a déjà été discuté: 1.5.3 on page 29.

[42]Frame Pointer

MOV écrit 0 dans W0. W0 est la partie basse 32 bits du registre 64-bit X0 :

Partie 32 bits haute	Partie 32 bits basse
X0	
	W0

Le résultat de la fonction est retourné via X0 et main renvoie 0, donc c'est ainsi que la valeur de retour est préparée. Mais pourquoi utiliser la partie 32-bit?

Parce que le type de donnée *int* en ARM64, tout comme en x86-64, est toujours 32-bit, pour une meilleure compatibilité.

Donc si la fonction renvoie un *int* 32-bit, seul les 32 premiers bits du registre X0 doivent être remplis.

Pour vérifier ceci, changeons un peu cet exemple et recompilons-le. Maintenant, main() renvoie une valeur sur 64-bit:

Listing 1.29 : main() renvoie une valeur de type uint64_t type

```
#include <stdio.h>
#include <stdint.h>

uint64_t main()
{
        printf ("Hello!\n");
        return 0;
}
```

Le résultat est le même, mais c'est à quoi ressemble MOV à cette ligne maintenant:

Listing 1.30 : GCC 4.8.1 sans optimisation + objdump

```
  4005a4:       d2800000        mov     x0, #0x0      // #0
```

LDP (*Load Pair*) remet les registres X29 et X30.

Il n'y a pas de point d'exclamation après l'instruction: celui signifie que les valeurs sont d'abord chargées depuis la pile, et ensuite SP est incrémenté de 16. Cela est appelé *post-index*.

Une nouvelle instruction est apparue en ARM64: RET. Elle fonctionne comme BX LR, un *hint* bit particulier est ajouté, qui informe le CPU qu'il s'agit d'un retour de fonction, et pas d'une autre instruction de saut, et il peut l'exécuter de manière plus optimale.

À cause de la simplicité de la fonction, GCC avec l'option d'optimisation génère le même code.

1.5.4 MIPS

Un mot à propos du «pointeur global»

Un concept MIPS important est le «pointeur global». Comme nous le savons déjà, chaque instruction MIPS a une taille de 32-bit, donc il est impossible d'avoir une adresse 32-bit dans une instruction: il faut pour cela utiliser une paire. (comme le

fait GCC dans notre exemple pour le chargement de l'adresse de la chaîne de texte). Il est possible, toutefois, de charger des données depuis une adresse dans l'intervalle $register - 32768...register + 32767$ en utilisant une seule instruction (car un offset signé de 16 bits peut être encodé dans une seule instruction). Nous pouvons alors allouer un registre dans ce but et dédier un bloc de 64KiB pour les données les plus utilisées. Ce registre dédié est appelé un «pointeur global» et il pointe au milieu du bloc de 64 KiB. Ce bloc contient en général les variables globales et les adresses des fonctions importées, comme `printf()`, car les développeurs de GCC ont décidé qu'obtenir l'adresse d'une fonction devait se faire en une instruction au lieu de deux. Dans un fichier ELF ce bloc de 64KiB se trouve en partie dans une section .sbss («small BSS[43]») pour les données non initialisées et .sdata («small data») pour celles initialisées. Cela implique que le programmeur peut choisir quelle donnée il/elle souhaite rendre accessible rapidement et doit les stocker dans .sdata/.sbss. Certains programmeurs old-school peuvent se souvenir du modèle de mémoire MS-DOS 11.7 on page 1297 ou des gestionnaires de mémoire MS-DOS comme XMS/EMS où toute la mémoire était divisée en bloc de 64KiB.

Ce concept n'est pas restreint à MIPS. Au moins les PowerPC utilisent aussi cette technique.

GCC avec optimisation

Considérons l'exemple suivant, qui illustre le concept de «pointeur global».

Listing 1.31 : GCC 4.4.5 avec optimisation (résultat en sortie de l'assembleur)

```
$LC0:
; \000 est l'octet à zéro en base octale:
        .ascii  "Hello, world!\012\000"
main:
; prologue de la fonction.
; définir GP:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-32
        addiu   $28,$28,%lo(__gnu_local_gp)
; sauver RA sur la pile locale:
        sw      $31,28($sp)
; charger l'adresse de la fonction puts() dans $25 depuis GP:
        lw      $25,%call16(puts)($28)
; charger l'adresse de la chaîne de texte dans $4 ($a0):
        lui     $4,%hi($LC0)
; sauter à puts(), en sauvant l'adresse de retour dans le register link:
        jalr    $25
        addiu   $4,$4,%lo($LC0) ; slot de retard de branchement
; restaurer RA:
        lw      $31,28($sp)
; copier 0 depuis $zero dans $v0:
        move    $2,$0
; retourner en sautant à la valeur dans RA:
        j       $31
; épilogue de la fonction:
```

[43] Block Started by Symbol

```
26          addiu   $sp,$sp,32 ; slot de retard de branchement + libérer la pile
    locale
```

Comme on le voit, le registre $GP est défini dans le prologue de la fonction pour pointer au milieu de ce bloc. Le registre RA est sauvé sur la pile locale. `puts()` est utilisé ici au lieu de `printf()`. L'adresse de la fonction `puts()` est chargée dans $25 en utilisant l'instruction `LW` («Load Word»). Ensuite l'adresse de la chaîne de texte est chargée dans $4 avec la paire d'instructions `LUI` ((«Load Upper Immediate») et `ADDIU` («Add Immediate Unsigned Word»). `LUI` défini les 16 bits de poids fort du registre (d'où le mot «upper» dans le nom de l'instruction) et `ADDIU` ajoute les 16 bits de poids faible de l'adresse.

`ADDIU` suit `JALR` (vous n'avez pas déjà oublié le *slot de délai de branchement*?). Le registre $4 est aussi appelé $A0, qui est utilisé pour passer le premier argument d'une fonction [44].

`JALR` («Jump and Link Register») saute à l'adresse stockée dans le registre $25 (adresse de `puts()`) en sauvant l'adresse de la prochaine instruction (LW) dans RA. C'est très similaire à ARM. Oh, encore une chose importante, l'adresse sauvée dans RA n'est pas l'adresse de l'instruction suivante (car c'est celle du *slot de délai* et elle est exécutée avant l'instruction de saut), mais l'adresse de l'instruction après la suivante (après le *slot de délai*). Par conséquent, $PC + 8$ est écrit dans RA pendant l'exécution de `JALR`, dans notre cas, c'est l'adresse de l'instruction `LW` après `ADDIU`.

`LW` («Load Word») à la ligne 20 restaure RA depuis la pile locale (cette instruction fait partie de l'épilogue de la fonction).

`MOVE` à la ligne 22 copie la valeur du registre $0 ($ZERO) dans $2 ($V0).

MIPS a un registre *constant*, qui contient toujours zéro. Apparemment, les développeurs de MIPS avaient à l'esprit que zéro est la constante la plus utilisée en programmation, utilisons donc le registre $0 à chaque fois que zéro est requis.

Un autre fait intéressant est qu'il manque en MIPS une instruction qui transfère des données entre des registres. En fait, `MOVE DST, SRC` est `ADD DST, SRC, $ZERO` ($DST = SRC + 0$), qui fait la même chose. Manifestement, les développeurs de MIPS voulaient une table des opcodes compacte. Cela ne signifie pas qu'il y a une addition à chaque instruction `MOVE`. Très probablement, le CPU optimise ces pseudo-instructions et l'UAL[45] n'est jamais utilisé.

`J` à la ligne 24 saute à l'adresse dans RA, qui effectue effectivement un retour de la fonction. `ADDIU` après `J` est en fait exécutée avant `J` (vous vous rappeler du *slot de délai de branchement*?) et fait partie de l'épilogue de la fonction. Voici un listing généré par IDA. Chaque registre a son propre pseudo nom:

Listing 1.32 : GCC 4.4.5 avec optimisation (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
```

[44] La table des registres MIPS est disponible en appendice .3.1 on page 1352

[45] Unité arithmétique et logique

```
6 ; prologue de la fonction.
7 ; définir GP:
8 .text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
9 .text:00000004                 addiu   $sp, -0x20
10 .text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
11 ; sauver RA sur la pile locale:
12 .text:0000000C                 sw      $ra, 0x20+var_4($sp)
13 ; sauver GP sur la pile locale:
14 ; pour une raison, cette instruction manque dans la sortie en assembleur de GCC:
15 .text:00000010                 sw      $gp, 0x20+var_10($sp)
16 ; charger l'adresse de la fonction puts() dans $9 depuis GP:
17 .text:00000014                 lw      $t9, (puts & 0xFFFF)($gp)
18 ; générer l'adresse de la chaîne de texte dans $a0:
19 .text:00000018                 lui     $a0, ($LC0 >> 16)  # "Hello, world!"
20 ; sauter à puts(), en sauvant l'adresse de retour dans le register link:
21 .text:0000001C                 jalr    $t9
22 .text:00000020                 la      $a0, ($LC0 & 0xFFFF)  # "Hello, world!"
23 ; restaurer RA:
24 .text:00000024                 lw      $ra, 0x20+var_4($sp)
25 ; copier 0 depuis $zero dans $v0:
26 .text:00000028                 move    $v0, $zero
27 ; retourner en sautant à la valeur dans RA:
28 .text:0000002C                 jr      $ra
29 ; épilogue de la fonction:
30 .text:00000030                 addiu   $sp, 0x20
```

L'instruction à la ligne 15 sauve la valeur de GP sur la pile locale, et cette instruction manque mystérieusement dans le listing de sortie de GCC, peut-être une erreur de GCC [46]. La valeur de GP doit effectivement être sauvée, car chaque fonction utilise sa propre fenêtre de 64KiB. Le registre contenant l'adresse de `puts()` est appelé $T9, car les registres préfixés avec T- sont appelés «temporaires» et leur contenu ne doit pas être préservé.

GCC sans optimisation

GCC sans optimisation est plus verbeux.

Listing 1.33 : GCC 4.4.5 sans optimisation (résultat en sortie de l'assembleur)

```
$LC0:
        .ascii  "Hello, world!\012\000"
main:
; prologue de la fonction.
; sauver RA ($31) et FP sur la pile:
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
        sw      $fp,24($sp)
; définir le pointeur de pile FP (stack frame pointer):
        move    $fp,$sp
```

[46] Apparemment, les fonctions générant les listings ne sont pas si critique pour les utilisateurs de GCC, donc des erreurs peuvent toujours subsister.

```
; définir GP:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
; charger l'adresse de la chaîne de texte:
        lui     $2,%hi($LC0)
        addiu   $4,$2,%lo($LC0)
; charger l'adresse de puts() en utilisant GP:
        lw      $2,%call16(puts)($28)
        nop
; appeler puts():
        move    $25,$2
        jalr    $25
        nop  ; slot de retard de branchement

; restaurer GP depuis la pile locale:
        lw      $28,16($fp)
; mettre le registre $2 ($V0) à zéro:
        move    $2,$0
; épilogue de la fonction.
; restaurer SP:
        move    $sp,$fp
; restaurer RA:
        lw      $31,28($sp)
; restaurer FP:
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
; sauter en RA:
        j       $31
        nop  ; slot de délai de branchement
```

Nous voyons ici que le registre FP est utilisé comme un pointeur sur la pile. Nous voyons aussi 3 NOPs. Le second et le troisième suivent une instruction de branchement. Peut-être que le compilateur GCC ajoute toujours des NOPs (à cause du *slot de retard de branchement*) après les instructions de branchement, et, si l'optimisation est demandée, il essaye alors de les éliminer. Donc, dans ce cas, ils sont laissés en place.

Voici le listing IDA :

Listing 1.34 : GCC 4.4.5 sans optimisation (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; prologue de la fonction.
; sauver RA et FP sur la pile:
.text:00000000                 addiu   $sp, -0x20
.text:00000004                 sw      $ra, 0x20+var_4($sp)
.text:00000008                 sw      $fp, 0x20+var_8($sp)
; définir FP (stack frame pointer):
.text:0000000C                 move    $fp, $sp
```

```
14 ; définir GP:
15 .text:00000010                  la      $gp, __gnu_local_gp
16 .text:00000018                  sw      $gp, 0x20+var_10($sp)
17 ; charger l'adresse de la chaîne de texte:
18 .text:0000001C                  lui     $v0, (aHelloWorld >> 16)  # "Hello, world!"
19 .text:00000020                  addiu   $a0, $v0, (aHelloWorld & 0xFFFF)  # "Hello, world!"
20 ; charger l'adresse de puts() en utilisant GP:
21 .text:00000024                  lw      $v0, (puts & 0xFFFF)($gp)
22 .text:00000028                  or      $at, $zero ; NOP
23 ; appeler puts():
24 .text:0000002C                  move    $t9, $v0
25 .text:00000030                  jalr    $t9
26 .text:00000034                  or      $at, $zero ; NOP
27 ; restaurer GP depuis la pile locale:
28 .text:00000038                  lw      $gp, 0x20+var_10($fp)
29 ; mettre le registre $2 ($V0) à zéro:
30 .text:0000003C                  move    $v0, $zero
31 ; épilogue de la fonction.
32 ; restaurer SP:
33 .text:00000040                  move    $sp, $fp
34 ; restaurer RA:
35 .text:00000044                  lw      $ra, 0x20+var_4($sp)
36 ; restaurer FP:
37 .text:00000048                  lw      $fp, 0x20+var_8($sp)
38 .text:0000004C                  addiu   $sp, 0x20
39 ; sauter en RA:
40 .text:00000050                  jr      $ra
41 .text:00000054                  or      $at, $zero ; NOP
```

Intéressant, IDA a reconnu les instructions LUI/ADDIU et les a agrégées en une pseudo instruction LA («Load Address») à la ligne 15. Nous pouvons voir que cette pseudo instruction a une taille de 8 octets! C'est une pseudo instruction (ou *macro*) car ce n'est pas une instruction MIPS réelle, mais plutôt un nom pratique pour une paire d'instructions.

Une autre chose est qu'IDA ne reconnaît pas les instructions NOP, donc ici elles se trouvent aux lignes 22, 26 et 41. C'est `OR $AT, $ZERO`. Essentiellement, cette instruction applique l'opération OR au contenu du registre $AT avec zéro, ce qui, bien sûr, est une instruction sans effet. MIPS, comme beaucoup d'autres ISAs, n'a pas une instruction NOP.

Rôle de la pile dans cet exemple

L'adresse de la chaîne de texte est passée dans le registre. Pourquoi définir une pile locale quand même? La raison de cela est que la valeur des registres RA et GP doit être sauvée quelque part (car `printf()` est appelée), et que la pile locale est utilisée pour cela. Si cela avait été une fonction leaf, il aurait été possible de se passer du prologue et de l'épilogue de la fonction, par exemple: 1.4.3 on page 11.

GCC avec optimisation : chargeons-le dans GDB

Listing 1.35 : extrait d'une session GDB

```
root@debian-mips:~# gcc hw.c -O3 -o hw

root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian
...
Reading symbols from /root/hw...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
Starting program: /root/hw

Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>:    lui     gp,0x42
0x00400644 <main+4>:    addiu   sp,sp,-32
0x00400648 <main+8>:    addiu   gp,gp,-30624
0x0040064c <main+12>:   sw      ra,28(sp)
0x00400650 <main+16>:   sw      gp,16(sp)
0x00400654 <main+20>:   lw      t9,-32716(gp)
0x00400658 <main+24>:   lui     a0,0x40
0x0040065c <main+28>:   jalr    t9
0x00400660 <main+32>:   addiu   a0,a0,2080
0x00400664 <main+36>:   lw      ra,28(sp)
0x00400668 <main+40>:   move    v0,zero
0x0040066c <main+44>:   jr      ra
0x00400670 <main+48>:   addiu   sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
0x2ab2de60 in printf () from /lib/libc.so.6
(gdb) x/s $a0
0x400820:        "hello, world"
(gdb)
```

1.5.5 Conclusion

La différence principale entre le code x86/ARM et x64/ARM64 est que le pointeur sur la chaîne a une taille de 64 bits. Le fait est que les CPUs modernes sont maintenant 64-bit à cause le la baisse du coût de la mémoire et du grand besoin de cette dernière par les applications modernes. Nous pouvons ajouter bien plus de mémoire à nos ordinateurs que les pointeurs 32-bit ne peuvent en adresser. Ainsi, tous les pointeurs sont maintenant 64-bit.

1.5.6 Exercices

- http://challenges.re/48
- http://challenges.re/49

1.6 Fonction prologue et épilogue

Un prologue de fonction est une séquence particulière d'instructions située au début d'une fonction. Il ressemble souvent à ce morceau de code:

```
    push    ebp
    mov     ebp, esp
    sub     esp, X
```

Ce que ces instructions font: sauvent la valeur du registre EBP dans la pile (push ebp), sauvent la valeur actuelle du registre ESP dans le registre EBP (mov ebp, esp) et enfin allouent de la mémoire dans la pile pour les variables locales de la fonction (sub esp, X).

La valeur du registre EBP reste la même durant la période où la fonction s'exécute et est utilisée pour accéder aux variables locales et aux arguments de la fonction.

Le registre ESP peut aussi être utilisé pour accéder aux variables locales et aux arguments de la fonction, cependant cette approche n'est pas pratique car sa valeur est susceptible de changer au cours de l'exécution de cette fonction.

L'épilogue de fonction libère la mémoire allouée dans la pile (mov esp, ebp), restaure l'ancienne valeur de EBP précédemment sauvegardée dans la pile (pop ebp) puis rend l'exécution à l'appelant (ret 0).

```
    mov    esp, ebp
    pop    ebp
    ret    0
```

Les prologues et épilogues de fonction sont généralement détectés par les désassembleurs pour déterminer où une fonction commence et où elle se termine.

1.6.1 Récursivité

Les prologues et épilogues de fonction peuvent affecter négativement les performances de la récursion.

Plus d'information sur la récursivité dans ce livre: 3.7.3 on page 614.

1.7 Une fonction vide: redux

Revenons sur l'exemple de la fonction vide 1.3 on page 8. Maintenant que nous connaissons le prologue et l'épilogue de fonction, ceci est une fonction vide 1.1 on page 8 compilée par GCC sans optimisation:

Listing 1.36 : GCC 8.2 x64 sans optimisation (résultat en sortie de l'assembleur)

```
f:
        push    rbp
        mov     rbp, rsp
        nop
        pop     rbp
        ret
```

C'est RET, mais le prologue et l'épilogue de la fonction, probablement, n'ont pas été optimisés et laissés tels quels. NOP semble être un autre artefact du compilateur. De toutes façons, la seule instruction effective ici est RET. Toutes les autres instructions peuvent être supprimées (ou optimisées).

1.8 Renvoyer des valeurs: redux

À nouveau, quand on connaît le prologue et l'épilogue de fonction, recompilons un exemple renvoyant une valeur (1.4 on page 10, 1.8 on page 10) en utilisant GCC sans optimisation:

Listing 1.37 : GCC 8.2 x64 sans optimisation (résultat en sortie de l'assembleur)

```
f:
        push    rbp
        mov     rbp, rsp
        mov     eax, 123
        pop     rbp
        ret
```

Les seules instructions efficaces ici sont MOV et RET, les autres sont – prologue et épilogue.

1.9 Pile

La pile est une des structures de données les plus fondamentales en informatique [47]. AKA[48] LIFO[49].

Techniquement, il s'agit d'un bloc de mémoire situé dans l'espace d'adressage d'un processus et qui est utilisé par le registre ESP en x86, RSP en x64 ou par le registre SP en ARM comme un pointeur dans ce bloc mémoire.

Les instructions d'accès à la pile sont PUSH et POP (en x86 ainsi qu'en ARM Thumb-mode). PUSH soustrait à ESP/RSP/SP 4 en mode 32-bit (ou 8 en mode 64-bit) et écrit ensuite le contenu de l'opérande associé à l'adresse mémoire pointée par ESP/RSP/SP.

[47] wikipedia.org/wiki/Call_stack
[48] Also Known As — Aussi connu sous le nom de
[49] Dernier entré, premier sorti

POP est l'opération inverse: elle récupère la donnée depuis l'adresse mémoire pointée par SP, l'écrit dans l'opérande associé (souvent un registre) puis ajoute 4 (ou 8) au pointeur de pile.

Après une allocation sur la pile, le pointeur de pile pointe sur le bas de la pile. PUSH décrémente le pointeur de pile et POP l'incrémente.

Le bas de la pile représente en réalité le début de la mémoire allouée pour le bloc de pile. Cela semble étrange, mais c'est comme ça.

ARM supporte à la fois les piles ascendantes et descendantes.

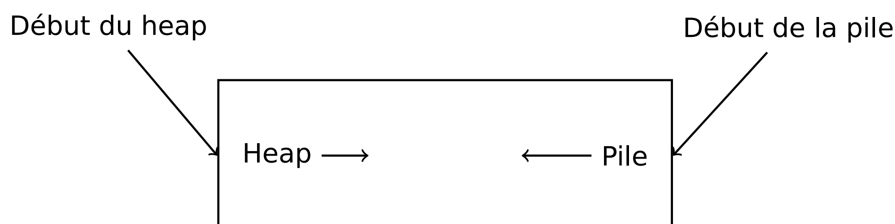
Par exemple les instructions STMFD/LDMFD, STMED[50]/LDMED[51] sont utilisées pour gérer les piles descendantes (qui grandissent vers le bas en commençant avec une adresse haute et évoluent vers une plus basse).

Les instructions STMFA[52]/LDMFA[53], STMEA[54]/LDMEA[55] sont utilisées pour gérer les piles montantes (qui grandissent vers les adresses hautes de l'espace d'adressage, en commençant avec une adresse située en bas de l'espace d'adressage).

1.9.1 Pourquoi la pile grandit en descendant ?

Intuitivement, on pourrait penser que la pile grandit vers le haut, i.e. vers des adresses plus élevées, comme n'importe qu'elle autre structure de données.

La raison pour laquelle la pile grandit vers le bas est probablement historique. Dans le passé, les ordinateurs étaient énormes et occupaient des pièces entières, il était facile de diviser la mémoire en deux parties, une pour le tas et une pour la pile. Évidemment, on ignorait quelle serait la taille du tas et de la pile durant l'exécution du programme, donc cette solution était la plus simple possible.



Dans [D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)][56]on peut lire:

> The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual

[50]Store Multiple Empty Descending (instruction ARM)
[51]Load Multiple Empty Descending (instruction ARM)
[52]Store Multiple Full Ascending (instruction ARM)
[53]Load Multiple Full Ascending (instruction ARM)
[54]Store Multiple Empty Ascending (instruction ARM)
[55]Load Multiple Empty Ascending (instruction ARM)
[56]Aussi disponible en URL

> address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a nonshared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a pile segment, which automatically grows downward as the hardware's pile pointer fluctuates.

Cela nous rappelle comment certains étudiants prennent des notes pour deux cours différents dans un seul et même cahier en prenant un cours d'un côté du cahier, et l'autre cours de l'autre côté. Les notes de cours finissent par se rencontrer à un moment dans le cahier quand il n'y a plus de place.

1.9.2 Quel est le rôle de la pile ?

Sauvegarder l'adresse de retour de la fonction

x86

Lorsque l'on appelle une fonction avec une instruction CALL, l'adresse du point exactement après cette dernière est sauvegardée sur la pile et un saut inconditionnel à l'adresse de l'opérande CALL est exécuté.

L'instruction CALL est équivalente à la paire d'instructions
`PUSH address_after_call / JMP operand`.

RET va chercher une valeur sur la pile et y saute —ce qui est équivalent à la paire d'instructions `POP tmp / JMP tmp`.

Déborder de la pile est très facile. Il suffit de lancer une récursion éternelle:

```
void f()
{
        f();
};
```

MSVC 2008 signale le problème:

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for ⤦
    ↳ 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths, ⤦
    ↳ function will cause runtime stack overflow
```

…mais génère tout de même le code correspondant:

```
?f@@YAXXZ PROC                   ; f
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ        ; f
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                   ; f
```

…Si nous utilisons l'option d'optimisation du compilateur (option /Ox) le code optimisé ne va pas déborder de la pile et au lieu de cela va fonctionner *correctemment*[57] :

```
?f@@YAXXZ PROC                   ; f
; Line 2
$LL3@f:
; Line 3
        jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                   ; f
```

GCC 4.4.1 génère un code similaire dans les deux cas, sans, toutefois émettre d'avertissement à propos de ce problème.

ARM

Les programmes ARM utilisent également la pile pour sauver les adresses de retour, mais différemment. Comme mentionné dans «Hello, world! » (1.5.3 on page 25), RA est sauvegardé dans LR (link register). Si l'on a toutefois besoin d'appeler une autre fonction et d'utiliser le registre LR une fois de plus, sa valeur doit être sauvegardée. Usuellement, cela se fait dans le prologue de la fonction.

Souvent, nous voyons des instructions comme `PUSH R4-R7,LR` en même temps que cette instruction dans l'épilogue `POP R4-R7,PC`—ces registres qui sont utilisés dans la fonction sont sauvegardés sur la pile, LR inclus.

Néanmoins, si une fonction n'appelle jamais d'autre fonction, dans la terminologie RISC elle est appelée *fonction leaf*[58]. Ceci a comme conséquence que les fonctions leaf ne sauvegardent pas le registre LR (car elles ne le modifient pas). Si une telle fonction est petite et utilise un petit nombre de registres, elle peut ne pas utiliser du tout la pile. Ainsi, il est possible d'appeler des fonctions leaf sans utiliser la pile. Ce qui peut être plus rapide sur des vieilles machines x86 car la mémoire externe n'est pas utilisée pour la pile [59]. Cela peut être utile pour des situations où la mémoire pour la pile n'est pas encore allouée ou disponible.

[57]ironique ici

[58]infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html

[59]Il y a quelque temps, sur PDP-11 et VAX, l'instruction CALL (appel d'autres fonctions) était coûteuse; jusqu'à 50% du temps d'exécution pouvait être passé à ça, il était donc considéré qu'avoir un grand nombre de petites fonctions était un anti-pattern [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].

Quelques exemples de fonctions leaf: 1.14.3 on page 140, 1.14.3 on page 141, 1.281 on page 405, 1.297 on page 428, 1.28.5 on page 428, 1.191 on page 274, 1.189 on page 271, 1.208 on page 295.

Passage des arguments d'une fonction

Le moyen le plus utilisé pour passer des arguments en x86 est appelé «cdecl » :

```
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12
```

La fonction appelée reçoit ses arguments par la pile.

Voici donc comment sont stockés les arguments sur la pile avant l'exécution de la première instruction de la fonction `f()` :

ESP	return address
ESP+4	argument#1, marqué dans IDA comme `arg_0`
ESP+8	argument#2, marqué dans IDA comme `arg_4`
ESP+0xC	argument#3, marqué dans IDA comme `arg_8`
…	…

Pour plus d'information sur les conventions d'appel, voir cette section (6.1 on page 953).

À propos, la fonction appelée n'a aucune d'information sur le nombre d'arguments qui ont été passés. Les fonctions C avec un nombre variable d'arguments (comme `printf()`) peuvent déterminer leur nombre en utilisant les spécificateurs de la chaîne de format (qui commencent pas le symbole %).

Si nous écrivons quelque comme:

```
printf("%d %d %d", 1234);
```

`printf()` va afficher 1234, et deux autres nombres aléatoires[60], qui sont situés à côté dans la pile.

C'est pourquoi la façon dont la fonction `main()` est déclarée n'est pas très importante: comme `main()`,
`main(int argc, char *argv[])` ou `main(int argc, char *argv[], char *envp[])`.

En fait, le code-CRT appelle `main()`, schématiquement, de cette façon:

```
push envp
push argv
push argc
call main
...
```

60Pas aléatoire dans le sens strict du terme, mais plutôt imprévisibles: **??** on page ??

Si vous déclarez main() comme main() sans argument, ils sont néanmoins toujours présents sur la pile, mais ne sont pas utilisés. Si vous déclarez main() as comme main(int argc, char *argv[]), vous pourrez utiliser les deux premiers arguments, et le troisième restera «invisible» pour votre fonction. Il est même possible de déclarer main() comme main(int argc), cela fonctionnera.

Un autre exemple apparenté: 6.1.10.

Autres façons de passer les arguments

Il est à noter que rien n'oblige les programmeurs à passer les arguments à travers la pile. Ce n'est pas une exigence. On peut implémenter n'importe quelle autre méthode sans utiliser du tout la pile.

Une méthode répandue chez les débutants en assembleur est de passer les arguments par des variables globales, comme:

Listing 1.38 : Code assembleur

```
        ...

        mov     X, 123
        mov     Y, 456
        call    do_something

        ...

X       dd      ?
Y       dd      ?

do_something proc near
        ; take X
        ; take Y
        ; do something
        retn
do_something endp
```

Mais cette méthode a un inconvénient évident: la fonction *do_something()* ne peut pas s'appeler elle-même récursivement (ou par une autre fonction), car il faudrait écraser ses propres arguments. La même histoire avec les variables locales: si vous les stockez dans des variables globales, la fonction ne peut pas s'appeler elle-même. Et ce n'est pas thread-safe [61]. Une méthode qui stocke ces informations sur la pile rend cela plus facile—elle peut contenir autant d'arguments de fonctions et/ou de valeurs, que la pile a d'espace.

[Donald E. Knuth, *The Art of Computer Programming*, Volume 1, 3rd ed., (1997), 189] mentionne un schéma encore plus étrange, particulièrement pratique sur les IBM System/360.

[61] Correctement implémenté, chaque thread aurait sa propre pile avec ses propres arguments/variables.

MS-DOS a une manière de passer tous les arguments de fonctions via des registres, par exemple, c'est un morceau de code pour un ancien MS-DOS 16-bit qui affiche “Hello, world!”:

```
mov  dx, msg      ; address of message
mov  ah, 9        ; 9 means "print string" function
int  21h          ; DOS "syscall"

mov  ah, 4ch      ; "terminate program" function
int  21h          ; DOS "syscall"

msg  db 'Hello, World!\$'
```

C'est presque similaire à la méthode 6.1.3 on page 955. Et c'est aussi très similaire aux appels systèmes sous Linux (6.3.1 on page 973) et Windows.

Si une fonction MS-DOS devait renvoyer une valeur booléenne (i.e., un simple bit, souvent pour indiquer un état d'erreur), le flag CF était souvent utilisé.

Par exemple:

```
mov ah, 3ch       ; create file
lea dx, filename
mov cl, 1
int 21h
jc  error
mov file_handle, ax
...
error:
...
```

En cas d'erreur, le flag CF est mis. Sinon, le handle du fichier nouvellement créé est retourné via AX.

Cette méthode est encore utilisée par les programmeurs en langage d'assemblage. Dans le code source de Windows Research Kernel (qui est très similaire à Windows 2003) nous pouvons trouver quelque chose comme ça (file *base/ntos/ke/i386/cpu.asm*) :

```
        public  Get386Stepping
Get386Stepping  proc

        call    MultiplyTest            ; Perform multiplication test
        jnc     short G3s00             ; if nc, muttest is ok
        mov     ax, 0
        ret
G3s00:
        call    Check386B0              ; Check for B0 stepping
        jnc     short G3s05             ; if nc, it's B1/later
        mov     ax, 100h                ; It is B0/earlier stepping
        ret

G3s05:
        call    Check386D1              ; Check for D1 stepping
        jc      short G3s10             ; if c, it is NOT D1
```

```
        mov     ax, 301h                ; It is D1/later stepping
        ret

G3s10:
        mov     ax, 101h                ; assume it is B1 stepping
        ret

        ...

MultiplyTest    proc

        xor     cx,cx                   ; 64K times is a nice round number
mlt00:  push    cx
        call    Multiply                ; does this chip's multiply work?
        pop     cx
        jc      short mltx              ; if c, No, exit
        loop    mlt00                   ; if nc, YEs, loop to try again
        clc
mltx:
        ret

MultiplyTest    endp
```

Stockage des variables locales

Une fonction peut allouer de l'espace sur la pile pour ses variables locales simplement en décrémentant le pointeur de pile vers le bas de la pile.

Donc, c'est très rapide, peu importe combien de variables locales sont définies. Ce n'est pas une nécessité de stocker les variables locales sur la pile. Vous pouvez les stocker où bon vous semble, mais c'est traditionnellement fait comme cela.

x86: alloca() function

Intéressons-nous à la fonction `alloca()` [62]

Cette fonction fonctionne comme `malloc()`, mais alloue de la mémoire directement sur la pile. L'espace de mémoire ne doit pas être libéré via un appel à la fonction `free()`, puisque l'épilogue de fonction (1.6 on page 41) remet ESP à son état initial ce qui va automatiquement libérer cet espace mémoire.

Intéressons-nous à l'implémentation d'`alloca()`. Cette fonction décale simplement ESP du nombre d'octets demandé vers le bas de la pile, faisant ESP sur le bloc *alloué*.

Essayons :

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
```

[62] Avec MSVC, l'implémentation de cette fonction peut être trouvée dans les fichiers `alloca16.asm` et `chkstk.asm` dans
`C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

```
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

La fonction `_snprintf()` fonctionne comme `printf()`, mais au lieu d'afficher le résultat sur la sortie standard (ex., dans un terminal ou une console), il l'écrit dans le buffer `buf`. La fonction `puts()` copie le contenu de `buf` dans la sortie standard. Évidemment, ces deux appels de fonctions peuvent être remplacés par un seul appel à la fonction `printf()`, mais nous devons illustrer l'utilisation de petit buffer.

MSVC

Compilons (MSVC 2010) :

Listing 1.39 : MSVC 2010

```
...

    mov     eax, 600  ; 00000258H
    call    __alloca_probe_16
    mov     esi, esp

    push    3
    push    2
    push    1
    push    OFFSET $SG2672
    push    600       ; 00000258H
    push    esi
    call    __snprintf

    push    esi
    call    _puts
    add     esp, 28

...
```

Le seul argument d'`alloca()` est passé via EAX (au lieu de le mettre sur la pile) [63].

[63]C'est parce que alloca() est plutôt une fonctionnalité intrinsèque du compilateur (11.4 on page 1290) qu'une fonction normale. Une des raisons pour laquelle nous avons besoin d'une fonction séparée au lieu

GCC + Syntaxe Intel

GCC 4.4.1 fait la même chose sans effectuer d'appel à des fonctions externes :

Listing 1.40 : GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 660
        lea     ebx, [esp+39]
        and     ebx, -16                ; align pointer by 16-byte border
        mov     DWORD PTR [esp], ebx    ; s
        mov     DWORD PTR [esp+20], 3
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
        mov     DWORD PTR [esp+4], 600  ; maxlen
        call    _snprintf
        mov     DWORD PTR [esp], ebx    ; s
        call    puts
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
```

GCC + Syntaxe AT&T

Voyons le même code mais avec la syntaxe AT&T :

Listing 1.41 : GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $660, %esp
        leal    39(%esp), %ebx
        andl    $-16, %ebx
        movl    %ebx, (%esp)
        movl    $3, 20(%esp)
        movl    $2, 16(%esp)
        movl    $1, 12(%esp)
        movl    $.LC0, 8(%esp)
```

de quelques instructions dans le code, et parce que l'implémentation d'alloca() par MSVC[64] a également du code qui lit depuis la mémoire récemment allouée pour laisser l'OS mapper la mémoire physique vers la VM[65]. Aprés l'appel à la fonction `alloca()`, ESP pointe sur un bloc de 600 octets que nous pouvons utiliser pour le tableau `buf`.

```
        movl    $600, 4(%esp)
        call    _snprintf
        movl    %ebx, (%esp)
        call    puts
        movl    -4(%ebp), %ebx
        leave
        ret
```

Le code est le même que le précédent.

Au fait, `movl $3, 20(%esp)` correspond à `mov DWORD PTR [esp+20], 3` avec la syntaxe intel. Dans la syntaxe AT&T, le format registre+offset pour l'adressage mémoire ressemble à `offset(%register)`.

(Windows) SEH

Les enregistrements SEH[66] sont aussi stockés dans la pile (s'ils sont présents). Lire à ce propos: (6.5.3 on page 996).

Protection contre les débordements de tampon

Lire à ce propos (1.26.2 on page 350).

Dé-allocation automatique de données dans la pile

Peut-être que la raison pour laquelle les variables locales et les enregistrements SEH sont stockés dans la pile est qu'ils sont automatiquement libérés quand la fonction se termine en utilisant simplement une instruction pour corriger la position du pointeur de pile (souvent `ADD`). Les arguments de fonction sont aussi désalloués automatiquement à la fin de la fonction. À l'inverse, toutes les données allouées sur le *heap* doivent être désallouées de façon explicite.

1.9.3 Une disposition typique de la pile

Une disposition typique de la pile dans un environnement 32-bit au début d'une fonction, avant l'exécution de sa première instruction ressemble à ceci:

…	…
ESP-0xC	variable locale#2, marqué dans IDA comme `var_8`
ESP-8	variable locale#1, marqué dans IDA comme `var_4`
ESP-4	valeur enregistrée deEBP
ESP	Adresse de retour
ESP+4	argument#1, marqué dans IDA comme `arg_0`
ESP+8	argument#2, marqué dans IDA comme `arg_4`
ESP+0xC	argument#3, marqué dans IDA comme `arg_8`
…	…

[66]Structured Exception Handling

1.9.4 Bruit dans la pile

> Quand quelqu'un dit que quelques chose est aléatoire, ce que cela signifie en pratique c'est qu'il n'est pas capable de voir les régularités de cette chose
>
> Stephen Wolfram, A New Kind of Science.

Dans ce livre les valeurs dites «bruitée» ou «poubelle» présente dans la pile ou dans la mémoire sont souvent mentionnées.

D'où viennent-elles? Ces valeurs ont été laissées sur la pile après l'exécution de fonctions précédentes. Par exemple:

```
#include <stdio.h>

void f1()
{
        int a=1, b=2, c=3;
};

void f2()
{
        int a, b, c;
        printf ("%d, %d, %d\n", a, b, c);
};

int main()
{
        f1();
        f2();
};
```

Compilons …

Listing 1.42 : sans optimisation MSVC 2010

```
$SG2752 DB      '%d, %d, %d', 0aH, 00H

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f1     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        mov     DWORD PTR _c$[ebp], 3
        mov     esp, ebp
        pop     ebp
        ret     0
_f1     ENDP
```

```
_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f2     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     eax, DWORD PTR _c$[ebp]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        push    OFFSET $SG2752 ; '%d, %d, %d'
        call    DWORD PTR __imp__printf
        add     esp, 16
        mov     esp, ebp
        pop     ebp
        ret     0
_f2     ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        call    _f1
        call    _f2
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Le compilateur va rouspéter un peu…

```
c:\Polygon\c>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for ↙
    ↳ 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

st.c
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'c' ↙
    ↳ used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'b' ↙
    ↳ used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'a' ↙
    ↳ used
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:st.exe
st.obj
```

Mais quand nous lançons le programme compilé …

```
c:\Polygon\c>st
1, 2, 3
```

Quel résultat étrange ! Aucune variables n'a été initialisées dans `f2()`. Ce sont des valeurs «fantômes» qui sont toujours dans la pile.

Chargeons cet exemple dans OllyDbg :

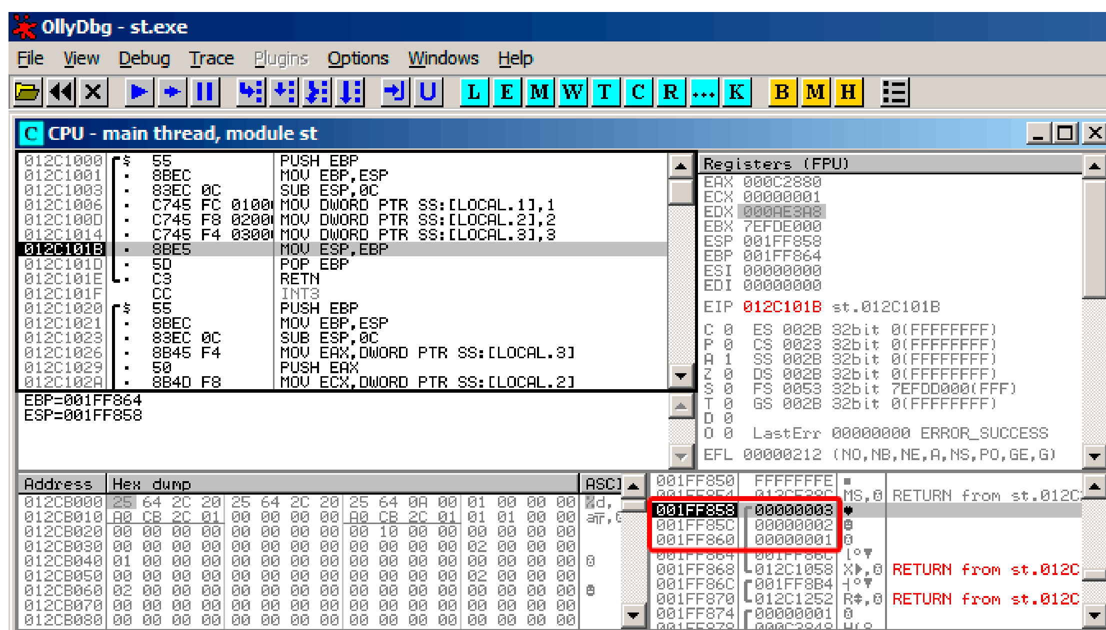


Fig. 1.6: OllyDbg : `f1()`

Quand `f1()` assigne les variable a, b et c, leurs valeurs sont stockées à l'adresse `0x1FF860` et ainsi de suite.

Et quand f2() s'exécute:

Fig. 1.7: OllyDbg : f2()

... a, b et c de la fonction f2() sont situées à la même adresse ! Aucunes autre fonction n'a encore écrasées ces valeurs, elles sont donc encore inchangées. Pour que cette situation arrive, il faut que plusieurs fonctions soit appelées les unes après les autres et que SP soit le même à chaque début de fonction (i.e., les fonctions doivent avoir le même nombre d'arguments). Les variables locales seront donc positionnées au même endroit dans la pile. Pour résumer, toutes les valeurs sur la pile sont des valeurs laissées par des appels de fonction précédents. Ces valeurs laissées sur la pile ne sont pas réellement aléatoires dans le sens strict du terme, mais elles sont imprévisibles. Y a t'il une autre option ? Il serait probablement possible de nettoyer des parties de la pile avant chaque nouvelle exécution de fonction, mais cela engendrerait du travail et du temps d'exécution (non nécessaire) en plus.

MSVC 2013

Cet exemple a été compilé avec MSVC 2010. Si vous essayez de compiler cet exemple avec MSVC 2013 et de l'exécuter, ces 3 nombres seront inversés:

```
c:\Polygon\c>st
3, 2, 1
```

Pourquoi ? J'ai aussi compilé cet exemple avec MSVC 2013 et constaté ceci:

Listing 1.43 : MSVC 2013

```
_a$ = -12       ; size = 4
_b$ = -8        ; size = 4
_c$ = -4        ; size = 4
_f2     PROC

...

_f2     ENDP

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f1     PROC

...

_f1     ENDP
```

Contrairement à MSVC 2010, MSVC 2013 alloue les variables a/b/c dans la fonction `f2()` dans l'ordre inverse puisqu'il se comporte différemment en raison d'un changement supposé dans son fonctionnement interne.Ceci est correct, car le standard du C/C++ n'a aucune règle sur l'ordre d'allocation des variables locales sur la pile.

1.9.5 Exercices

- http://challenges.re/51
- http://challenges.re/52

1.10 Fonction presque vide

Ceci est un morceau de code réel que j'ai trouvé dans Boolector[67] :

```
// forward declaration. the function is residing in some other module:
int boolector_main (int argc, char **argv);

// executable
int main (int argc, char **argv)
{
        return boolector_main (argc, argv);
}
```

Pourquoi quelqu'un ferait-il comme ça? Je ne sais pas mais mon hypothèse est que `boolector_main()` peut être compilée dans une sorte de DLL ou bibliothèque dynamique, et appelée depuis une suite de test. Certainement qu'une suite de test peut préparer les variables `argc/argv` comme le ferait CRT.

Il est intéressant de voir comment c'est compilé:

[67] https://boolector.github.io/

Listing 1.44 : GCC 8.2 x64 sans optimisation (résultat en sortie de l'assembleur)

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR -4[rbp], edi
        mov     QWORD PTR -16[rbp], rsi
        mov     rdx, QWORD PTR -16[rbp]
        mov     eax, DWORD PTR -4[rbp]
        mov     rsi, rdx
        mov     edi, eax
        call    boolector_main
        leave
        ret
```

Ceci est OK, le prologue (non optimisé) déplace inutilement deux arguments, CALL, épilogue, RET. Mais regardons la version optimisée:

Listing 1.45 : GCC 8.2 x64 avec optimisation (résultat en sortie de l'assembleur)

```
main:
        jmp     boolector_main
```

Aussi simple que ça: la pile et les registres ne sont pas touchés et `boolector_main()` a le même ensemble d'arguments. Donc, tout ce que nous avons à faire est de passer l'exécution à une autre adresse.

Ceci est proche d'une fonction thunk.

Nous verons queelque chose de plus avancé plus tard: 1.11.2 on page 75, 1.21.1 on page 205.

1.11 printf() avec plusieurs arguments

Maintenant, améliorons l'exemple *Hello, world!* (1.5 on page 12) en remplaçant `printf()` dans la fonction `main()` par ceci:

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
        return 0;
};
```

1.11.1 x86

x86: 3 arguments entier

MSVC

En le compilant avec MSVC 2010 Express nous obtenons:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H

...

        push    3
        push    2
        push    1
        push    OFFSET $SG3830
        call    _printf
        add     esp, 16                                 ; 00000010H
```

Presque la même chose, mais maintenant nous voyons que les arguments de `printf()` sont poussés sur la pile en ordre inverse. Le premier argument est poussé en dernier.

À propos, dans un environnement 32-bit les variables de type *int* ont une taille de 32-bit. ce qui fait 4 octets.

Donc, nous avons 4 arguments ici. $4 * 4 = 16$ —ils occupent exactement 16 octets dans la pile: un pointeur 32-bit sur une chaîne et 3 nombres de type *int*.

Lorsque le pointeur de pile (registre ESP) est re-modifié par l'instruction `ADD ESP, X` après un appel de fonction, souvent, le nombre d'arguments de la fonction peut-être déduit en divisant simplement X par 4.

Bien sûr, cela est spécifique à la convention d'appel *cdecl*, et seulement pour un environnement 32-bit.

Voir aussi la section sur les conventions d'appel (6.1 on page 953).

Dans certains cas, plusieurs fonctions se terminent les une après les autres, le compilateur peut concaténer plusieurs instructions «ADD ESP, X» en une seule, après le dernier appel:

```
push a1
push a2
call ...
...
push a1
call ...
...
push a1
push a2
push a3
call ...
add esp, 24
```

Voici un exemple réel:

Listing 1.46 : x86

```
.text:100113E7    push    3
.text:100113E9    call    sub_100018B0 ; prendre un argument (3)
.text:100113EE    call    sub_100019D0 ; ne prendre aucun argument
.text:100113F3    call    sub_10006A90 ; ne prendre aucun argument
```

```
.text:100113F8    push    1
.text:100113FA    call    sub_100018B0 ; prendre un argument (1)
.text:100113FF    add     esp, 8       ; supprimer deux arguments de la pile
    à la fois
```

MSVC et OllyDbg

Maintenant, essayons de charger cet exemple dans OllyDbg. C'est l'un des débuggers en espace utilisateur win32 les plus populaire. Nous pouvons compiler notre exemple avec l'option /MD de MSVC 2012, qui signifie lier avec MSVCR*.DLL, ainsi nous verrons clairement les fonctions importées dans le debugger.

Ensuite chargeons l'exécutable dans OllyDbg. Le tout premier point d'arrêt est dans ntdll.dll, appuyez sur F9 (run). Le second point d'arrêt est dans le code CRT. Nous devons maintenant trouver la fonction main().

Trouvez ce code en vous déplaçant au tout début du code (MSVC alloue la fonction main() au tout début de la section de code) :

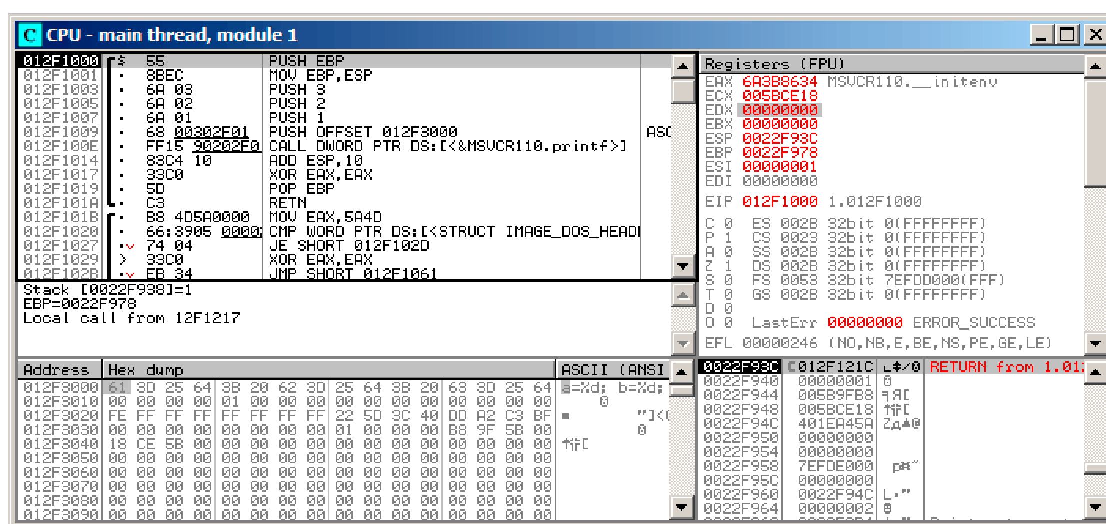


Fig. 1.8: OllyDbg : le tout début de la fonction main()

Clickez sur l'instruction PUSH EBP, pressez F2 (mettre un point d'arrêt) et pressez F9 (lancer le programme). Nous devons effectuer ces actions pour éviter le code CRT, car il ne nous intéresse pas pour le moment.

Presser F8 (enjamber) 6 fois, i.e. sauter 6 instructions:

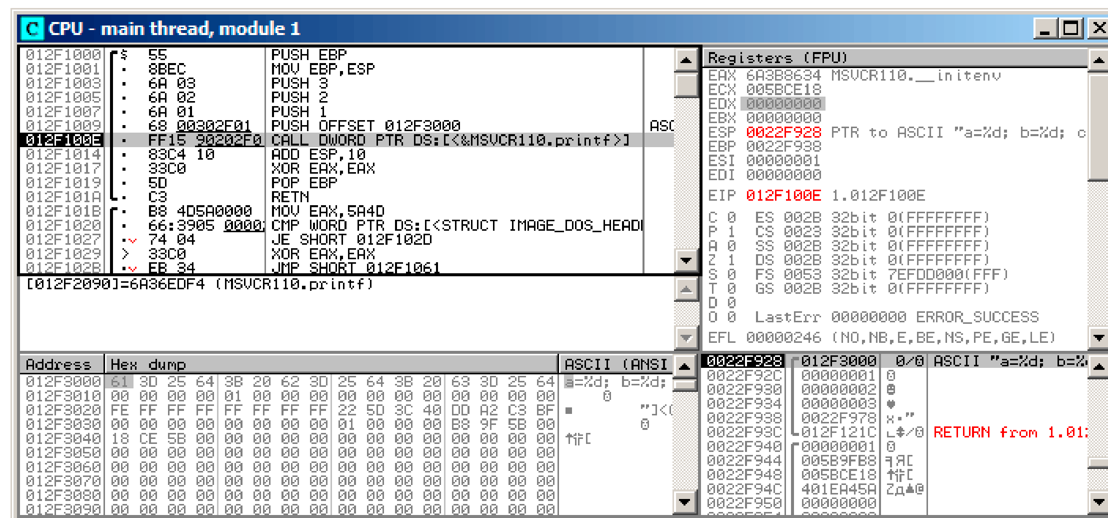


Fig. 1.9: OllyDbg : avant l'exécution de `printf()`

Maintenant le PC pointe vers l'instruction `CALL printf`. OllyDbg, comme d'autres debuggers, souligne la valeur des registres qui ont changé. Donc, chaque fois que vous appuyez sur F8, EIP change et sa valeur est affichée en rouge. ESP change aussi, car les valeurs des arguments sont poussées sur la pile.

Où sont les valeurs dans la pile? Regardez en bas à droite de la fenêtre du debugger:

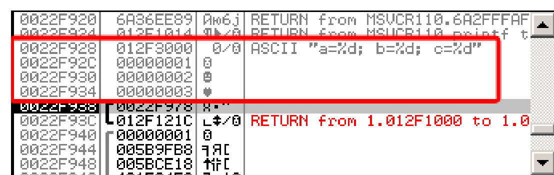


Fig. 1.10: OllyDbg : pile après que les valeurs des arguments aient été poussées (Le rectangle rouge a été ajouté par l'auteur dans un éditeur graphique)

Nous pouvons voir 3 colonnes ici: adresse dans la pile, valeur dans la pile et quelques commentaires additionnels d'OllyDbg. OllyDbg peut détecter les pointeurs sur des chaînes ASCII dans la pile, donc il rapporte la chaîne de `printf()` ici.

Il est possible de faire un clic-droit sur la chaîne de format, cliquer sur «Follow in dump», et la chaîne de format va apparaître dans la fenêtre en bas à gauche du debugger. qui affiche toujours des parties de la mémoire. Ces valeurs en mémoire peuvent être modifiées. Il est possible de changer la chaîne de format, auquel cas le résultat de notre exemple sera différent. Cela n'est pas très utile dans le cas présent,

mais ce peut-être un bon exercice pour commencer à comprendre comment tout fonctionne ici.

Appuyer sur F8 (enjamber).

Nous voyons la sortie suivante dans la console:

```
a=1; b=2; c=3
```

Regardons comment les registres et la pile ont changés:

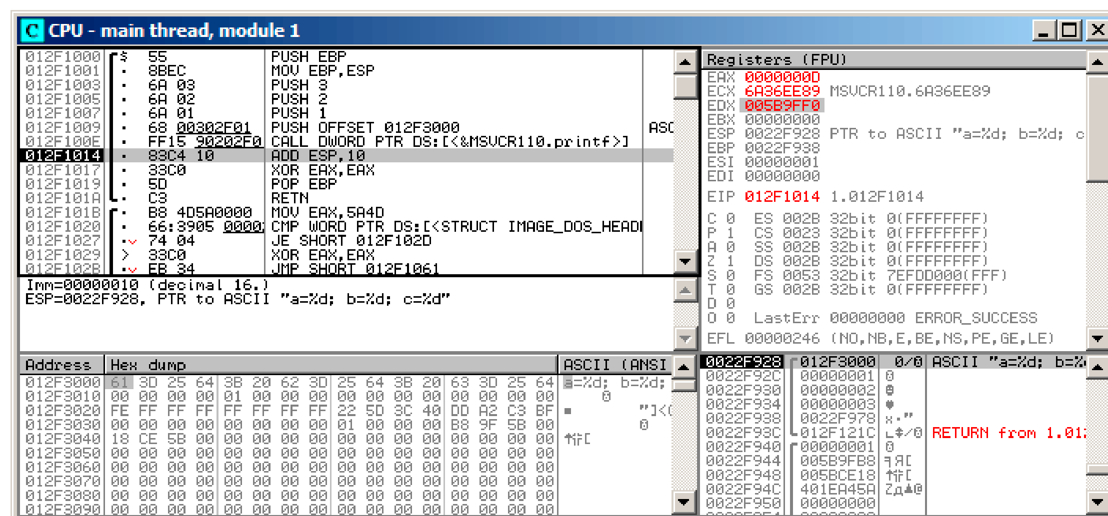


Fig. 1.11: OllyDbg après l'exécution de `printf()`

Le registre EAX contient maintenant `0xD` (13). C'est correct, puisque `printf()` renvoie le nombre de caractères écrits. La valeur de EIP a changé: en effet, il contient maintenant l'adresse de l'instruction venant après `CALL printf`. Les valeurs de ECX et EDX ont également changé. Apparemment, le mécanisme interne de la fonction `printf()` les a utilisés pour dans ses propres besoins.

Un fait très important est que ni la valeur de ESP, ni l'état de la pile n'ont été changés! Nous voyons clairement que la chaîne de format et les trois valeurs correspondantes sont toujours là. C'est en effet le comportement de la convention d'appel *cdecl* : l'appelée ne doit pas remettre ESP à sa valeur précédente. L'appelant est responsable de le faire.

Appuyer sur F8 à nouveau pour exécuter l'instruction ADD ESP, 10 :

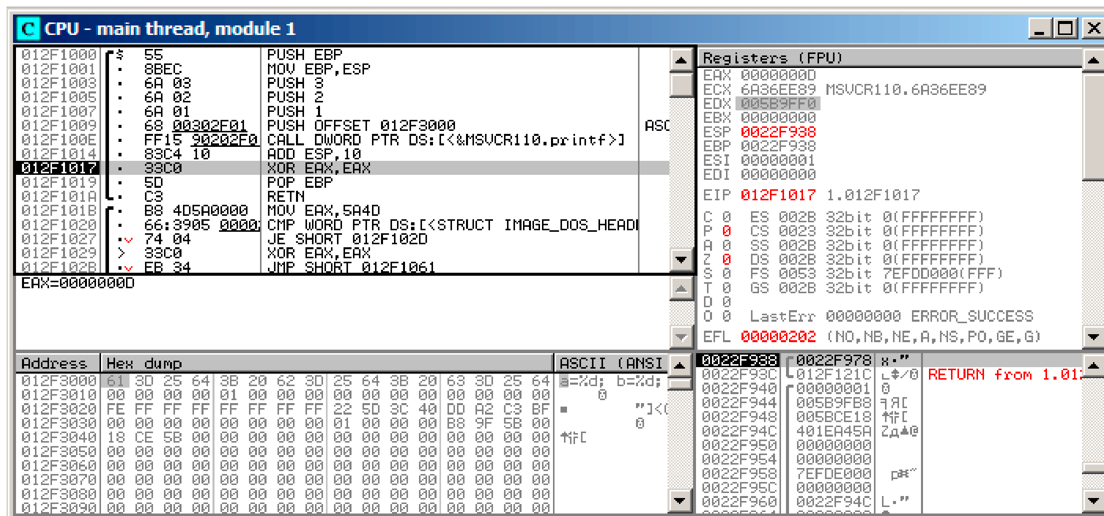


Fig. 1.12: OllyDbg : après l'exécution de l'instruction ADD ESP, 10

ESP a changé, mais les valeurs sont toujours dans la pile! Oui, bien sûr; il n'y a pas besoin de mettre ces valeurs à zéro ou quelque chose comme ça. Tout ce qui se trouve au dessus du pointeur de pile (SP) est du *bruit* ou des *déchets* et n'a pas du tout de signification. Ça prendrait beaucoup de temps de mettre à zéro les entrées inutilisées de la pile, et personne n'a vraiment besoin de le faire.

GCC

Maintenant, compilons la même programme sous Linux en utilisant GCC 4.4.1 et regardons ce que nous obtenons dans IDA :

```
main            proc near

var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
                mov     [esp+10h+var_4], 3
                mov     [esp+10h+var_8], 2
                mov     [esp+10h+var_C], 1
                mov     [esp+10h+var_10], eax
```

```
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Il est visible que la différence entre le code MSVC et celui de GCC est seulement dans la manière dont les arguments sont stockés sur la pile. Ici GCC manipule directement la pile sans utiliser PUSH/POP.

GCC et GDB

Essayons cet exemple dans GDB[68] sous Linux.

L'option -g indique au compilateur d'inclure les informations de debug dans le fichier exécutable.

```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/1...done.
```

Listing 1.47 : let's set breakpoint on printf()

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Lançons le programme. Nous n'avons pas la code source de la fonction printf() ici, donc GDB ne peut pas le montrer, mais pourrait.

```
(gdb) run
Starting program: /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
29      printf.c: No such file or directory.
```

Afficher 10 éléments de la pile. La colonne la plus à gauche contient les adresses de la pile.

```
(gdb) x/10w $esp
0xbffff11c:     0x0804844a      0x080484f0      0x00000001      0x00000002
0xbffff12c:     0x00000003      0x08048460      0x00000000      0x00000000
0xbffff13c:     0xb7e29905      0x00000001
```

Le tout premier élément est la RA (0x0804844a). Nous pouvons le vérifier en désassemblant la mémoire à cette adresse:

[68] GNU Debugger

```
(gdb) x/5i 0x0804844a
   0x804844a <main+45>: mov    $0x0,%eax
   0x804844f <main+50>: leave
   0x8048450 <main+51>: ret
   0x8048451:   xchg   %ax,%ax
   0x8048453:   xchg   %ax,%ax
```

Les deux instructions XCHG sont des instructions sans effet, analogues à NOPs.

Le second élément (`0x080484f0`) est l'adresse de la chaîne de format:

```
(gdb) x/s 0x080484f0
0x80484f0:      "a=%d; b=%d; c=%d"
```

Les 3 éléments suivants (1, 2, 3) sont les arguments de `printf()`. Le reste des éléments sont juste des «restes» sur la pile, mais peuvent aussi être des valeurs d'autres fonctions, leurs variables locales, etc. Nous pouvons les ignorer pour le moment.

Lancer la commande «finish». Cette commande ordonne à GDB d'«exécuter toutes les instructions jusqu'à la fin de la fonction». Dans ce cas: exécuter jusqu'à la fin de `printf()`.

```
(gdb) finish
Run till exit from #0  __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at ↙
    ↘ printf.c:29
main () at 1.c:6
6               return 0;
Value returned is $2 = 13
```

GDB montre ce que `printf()` a renvoyé dans EAX (13). C'est le nombre de caractères écrits, exactement comme dans l'exemple avec OllyDbg.

Nous voyons également «return 0;» et l'information que cette expression se trouve à la ligne 6 du fichier `1.c`. En effet, le fichier `1.c` se trouve dans le répertoire courant, et GDB y a trouvé la chaîne. Comment est-ce que GDB sait quelle ligne est exécutée à un instant donné? Cela est du au fait que lorsque le compilateur génère les informations de debug, il sauve également une table contenant la relation entre le numéro des lignes du code source et les adresses des instructions. GDB est un debugger niveau source, après tout.

Examinons les registres. 13 in EAX :

```
(gdb) info registers
eax            0xd      13
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844a        0x804844a <main+45>
```

```
...
```

Désassemblons les instructions courantes. La flèche pointe sur la prochaine instruction qui sera exécutée.

```
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   %ebp
   0x0804841e <+1>:     mov    %esp,%ebp
   0x08048420 <+3>:     and    $0xfffffff0,%esp
   0x08048423 <+6>:     sub    $0x10,%esp
   0x08048426 <+9>:     movl   $0x3,0xc(%esp)
   0x0804842e <+17>:    movl   $0x2,0x8(%esp)
   0x08048436 <+25>:    movl   $0x1,0x4(%esp)
   0x0804843e <+33>:    movl   $0x80484f0,(%esp)
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    $0x0,%eax
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

GDB utilise la syntaxe AT&T par défaut. Mais il est possible de choisir la syntaxe Intel:

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   ebp
   0x0804841e <+1>:     mov    ebp,esp
   0x08048420 <+3>:     and    esp,0xfffffff0
   0x08048423 <+6>:     sub    esp,0x10
   0x08048426 <+9>:     mov    DWORD PTR [esp+0xc],0x3
   0x0804842e <+17>:    mov    DWORD PTR [esp+0x8],0x2
   0x08048436 <+25>:    mov    DWORD PTR [esp+0x4],0x1
   0x0804843e <+33>:    mov    DWORD PTR [esp],0x80484f0
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    eax,0x0
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

Exécuter la ligne suivante de code C/C++. GDB montre une parenthèse fermante, signifiant la fin du bloc.

```
(gdb) step
7       };
```

Examinons les registres après l'exécution de l'instruction `MOV EAX, 0`. En effet, EAX est à zéro à ce stade.

```
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
```

```
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844f        0x804844f <main+50>
...
```

x64: 8 arguments entier

Pour voir comment les autres arguments sont passés par la pile, changeons encore notre exemple en augmentant le nombre d'arguments à 9 (chaîne de format de `printf()` + 8 variables *int*) :

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3,⤦
    ↳  4, 5, 6, 7, 8);
        return 0;
};
```

MSVC

Comme il a déjà été mentionné, les 4 premiers arguments sont passés par les registres RCX, RDX, R8, R9 sous Win64, tandis les autres le sont—par la pile. C'est exactement de que l'on voit ici. Toutefois, l'instruction `MOV` est utilisée ici à la place de PUSH, donc les valeurs sont stockées sur la pile d'une manière simple.

Listing 1.48 : MSVC 2012 x64

```
$SG2923 DB      'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d', 0aH, 00H

main    PROC
        sub     rsp, 88

        mov     DWORD PTR [rsp+64], 8
        mov     DWORD PTR [rsp+56], 7
        mov     DWORD PTR [rsp+48], 6
        mov     DWORD PTR [rsp+40], 5
        mov     DWORD PTR [rsp+32], 4
        mov     r9d, 3
        mov     r8d, 2
        mov     edx, 1
        lea     rcx, OFFSET FLAT:$SG2923
        call    printf

        ; renvoyer 0
        xor     eax, eax
```

```
        add     rsp, 88
        ret     0
main    ENDP
_TEXT   ENDS
END
```

Le lecteur observateur pourrait demander pourquoi 8 octets sont alloués sur la pile pour les valeurs *int*, alors que 4 suffisent? Oui, il faut se rappeler: 8 octets sont alloués pour tout type de données plus petit que 64 bits. Ceci est instauré pour des raisons de commodités: cela rend facile le calcul de l'adresse de n'importe quel argument. En outre, ils sont tous situés à des adresses mémoires alignées. Il en est de même dans les environnements 32-bit: 4 octets sont réservés pour tout types de données.

GCC

Le tableau est similaire pour les OS x86-64 *NIX, excepté que les 6 premiers arguments sont passés par les registres RDI, RSI, RDX, RCX, R8, R9. Tout les autres—par la pile. GCC génère du code stockant le pointeur de chaîne dans EDI au lieu de RDI—nous l'avons noté précédemment: 1.5.2 on page 22.

Nous avions également noté que le registre EAX a été vidé avant l'appel à `printf()` : 1.5.2 on page 22.

Listing 1.49 : GCC 4.4.6 x64 avec optimisation

```
.LC0:
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"

main:
        sub     rsp, 40

        mov     r9d, 5
        mov     r8d, 4
        mov     ecx, 3
        mov     edx, 2
        mov     esi, 1
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax ; nombre de registres vectoriels
        mov     DWORD PTR [rsp+16], 8
        mov     DWORD PTR [rsp+8], 7
        mov     DWORD PTR [rsp], 6
        call    printf

        ; renvoyer 0

        xor     eax, eax
        add     rsp, 40
        ret
```

GCC + GDB

Essayons cet exemple dans GDB.

```
$ gcc -g 2.c -o 2
```

```
$ gdb 2
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/2...done.
```

Listing 1.50 : mettons le point d'arrêt à `printf()`, et lançons

```
(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program: /home/dennis/polygon/2

Breakpoint 1, __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d ⤦
    ↪ ; g=%d; h=%d\n") at printf.c:29
29      printf.c: No such file or directory.
```

Les registres RSI/RDX/RCX/R8/R9 ont les valeurs attendues. RIP contient l'adresse de la toute première instruction de la fonction `printf()`.

```
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x3      3
rdx            0x2      2
rsi            0x1      1
rdi            0x400628 4195880
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf38   0x7fffffffdf38
r8             0x4      4
r9             0x5      5
r10            0x7fffffffdce0   140737488346336
r11            0x7ffff7a65f60   140737348263776
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x7ffff7a65f60   0x7ffff7a65f60 <__printf>
...
```

Listing 1.51 : inspectons la chaîne de format

```
(gdb) x/s $rdi
0x400628:       "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
```

Affichons la pile avec la commande x/g cette fois—*g* est l'unité pour *giant words*, i.e., mots de 64-bit.

```
(gdb) x/10g $rsp
0x7fffffffdf38: 0x0000000000400576      0x0000000000000006
0x7fffffffdf48: 0x0000000000000007      0x00007fff00000008
0x7fffffffdf58: 0x0000000000000000      0x0000000000000000
0x7fffffffdf68: 0x00007ffff7a33de5      0x0000000000000000
0x7fffffffdf78: 0x00007fffffffe048      0x0000000100000000
```

Le tout premier élément de la pile, comme dans le cas précédent, est la RA. 3 valeurs sont aussi passées par la pile: 6, 7, 8. Nous voyons également que 8 est passé avec les 32-bits de poids fort non effacés: `0x00007fff00000008`. C'est en ordre, car les valeurs sont d'un type *int*, qui est 32-bit. Donc, la partie haute du registre ou l'élément de la pile peuvent contenir des «restes de données aléatoires».

Si vous regardez où le contrôle reviendra après l'exécution de `printf()`, GDB affiche la fonction `main()` en entier:

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x0000000000400576
Dump of assembler code for function main:
   0x000000000040052d <+0>:     push   rbp
   0x000000000040052e <+1>:     mov    rbp,rsp
   0x0000000000400531 <+4>:     sub    rsp,0x20
   0x0000000000400535 <+8>:     mov    DWORD PTR [rsp+0x10],0x8
   0x000000000040053d <+16>:    mov    DWORD PTR [rsp+0x8],0x7
   0x0000000000400545 <+24>:    mov    DWORD PTR [rsp],0x6
   0x000000000040054c <+31>:    mov    r9d,0x5
   0x0000000000400552 <+37>:    mov    r8d,0x4
   0x0000000000400558 <+43>:    mov    ecx,0x3
   0x000000000040055d <+48>:    mov    edx,0x2
   0x0000000000400562 <+53>:    mov    esi,0x1
   0x0000000000400567 <+58>:    mov    edi,0x400628
   0x000000000040056c <+63>:    mov    eax,0x0
   0x0000000000400571 <+68>:    call   0x400410 <printf@plt>
   0x0000000000400576 <+73>:    mov    eax,0x0
   0x000000000040057b <+78>:    leave
   0x000000000040057c <+79>:    ret
End of assembler dump.
```

Laissons se terminer l'exécution de `printf()`, exécutez l'instruction mettant EAX à zéro, et notez que le registre EAX à une valeur d'exactement zéro. RIP pointe maintenant sur l'instruction LEAVE, i.e, la pénultième de la fonction `main()`.

```
(gdb) finish
Run till exit from #0  __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e⤦
    ↳ =%d; f=%d; g=%d; h=%d\n") at printf.c:29
a=1; b=2; c=3; d=4; e=5; f=6; g=7; h=8
main () at 2.c:6
6               return 0;
Value returned is $1 = 39
(gdb) next
7       };
(gdb) info registers
rax            0x0      0
```

```
rbx            0x0      0
rcx            0x26     38
rdx            0x7ffff7dd59f0   140737351866864
rsi            0x7fffffd9       2147483609
rdi            0x0      0
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf40   0x7fffffffdf40
r8             0x7ffff7dd26a0   140737351853728
r9             0x7ffff7a60134   140737348239668
r10            0x7fffffffd5b0   140737488344496
r11            0x7ffff7a95900   140737348458752
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x40057b 0x40057b <main+78>
...
```

1.11.2 ARM

ARM: 3 arguments entier

Le schéma ARM traditionnel pour passer des arguments (convention d'appel) se comporte de cette façon: les 4 premiers arguments sont passés par les registres R0-R3; les autres par la pile. Cela ressemble au schéma de passage des arguments dans fastcall (6.1.3 on page 955) ou win64 (6.1.5 on page 957).

ARM 32-bit

sans optimisation Keil 6/2013 (Mode ARM)

Listing 1.52 : sans optimisation Keil 6/2013 (Mode ARM)

```
.text:00000000 main
.text:00000000 10 40 2D E9   STMFD   SP!, {R4,LR}
.text:00000004 03 30 A0 E3   MOV     R3, #3
.text:00000008 02 20 A0 E3   MOV     R2, #2
.text:0000000C 01 10 A0 E3   MOV     R1, #1
.text:00000010 08 00 8F E2   ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d"
.text:00000014 06 00 00 EB   BL      __2printf
.text:00000018 00 00 A0 E3   MOV     R0, #0          ; renvoyer 0
.text:0000001C 10 80 BD E8   LDMFD   SP!, {R4,PC}
```

Donc, les 4 premiers arguments sont passés par les registres R0-R3 dans cet ordre: un pointeur sur la chaîne de format de `printf()` dans R0, puis 1 dans R1, 2 dans R2 et 3 dans R3. L'instruction en 0x18 écrit 0 dans R0—c'est la déclaration C de *return 0*.

avec optimisation Keil 6/2013 génère le même code.

avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.53 : avec optimisation Keil 6/2013 (Mode Thumb)

```
.text:00000000 main
.text:00000000 10 B5        PUSH    {R4,LR}
.text:00000002 03 23        MOVS    R3, #3
.text:00000004 02 22        MOVS    R2, #2
.text:00000006 01 21        MOVS    R1, #1
.text:00000008 02 A0        ADR     R0, aADBDCD      ; "a=%d; b=%d; c=%d"
.text:0000000A 00 F0 0D F8  BL      __2printf
.text:0000000E 00 20        MOVS    R0, #0
.text:00000010 10 BD        POP     {R4,PC}
```

Il n'y a pas de différence significative avec le code non optimisé pour le mode ARM.

avec optimisation Keil 6/2013 (Mode ARM) + supprimons le retour

Retravaillons légèrement l'exemple en supprimant *return 0* :

```
#include <stdio.h>

void main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
};
```

Le résultat est quelque peu inhabituel:

Listing 1.54 : avec optimisation Keil 6/2013 (Mode ARM)

```
.text:00000014 main
.text:00000014 03 30 A0 E3   MOV     R3, #3
.text:00000018 02 20 A0 E3   MOV     R2, #2
.text:0000001C 01 10 A0 E3   MOV     R1, #1
.text:00000020 1E 0E 8F E2   ADR     R0, aADBDCD      ; "a=%d; b=%d; c=%d\n"
.text:00000024 CB 18 00 EA   B       __2printf
```

C'est la version optimisée (-O3) pour le mode ARM et cette fois nous voyons B comme dernière instruction au lieu du BL habituel. Une autre différence entre cette version optimisée et la précédente (compilée sans optimisation) est l'absence de fonctions prologue et épilogue (les instructions qui préservent les valeurs des registres R0 et LR). L'instruction B saute simplement à une autre adresse, sans manipuler le registre LR, de façon similaire au JMP en x86. Pourquoi est-ce que fonctionne? Parce ce code est en fait bien équivalent au précédent. Il y a deux raisons principales: 1) Ni la pile ni SP (pointeur de pile) ne sont modifiés; 2) l'appel à `printf()` est la dernière instruction, donc il ne se passe rien après. A la fin, la fonction `printf()` rend simplement le contrôle à l'adresse stockée dans LR. Puisque LR contient actuellement l'adresse du point depuis lequel notre fonction a été appelée alors le contrôle après `printf()` sera redonné à ce point. Par conséquent, nous n'avons pas besoin de sauver LR car

il ne nous est pas nécessaire de le modifier. Et il ne nous est non plus pas nécessaire de modifier LR car il n'y a pas d'autre appel de fonction excepté `printf()`. Par ailleurs, après cet appel nous ne faisons rien d'autre! C'est la raison pour laquelle une telle optimisation est possible.

Cette optimisation est souvent utilisée dans les fonctions où la dernière déclaration est un appel à une autre fonction. Un exemple similaire est présenté ici: 1.21.1 on page 206.

Un cas un peu plus simple a été décrit plus haut: 1.10 on page 58.

ARM64

GCC (Linaro) 4.9 sans optimisation

Listing 1.55 : GCC (Linaro) 4.9 sans optimisation

```
.LC1:
        .string "a=%d; b=%d; c=%d"
f2:
; sauver FP et LR sur la pile:
        stp     x29, x30, [sp, -16]!
; définir la pile (FP=SP):
        add     x29, sp, 0
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
        mov     w1, 1
        mov     w2, 2
        mov     w3, 3
        bl      printf
        mov     w0, 0
; restaurer FP et LR
        ldp     x29, x30, [sp], 16
        ret
```

La première instruction STP (*Store Pair*) sauve FP (X29) et LR (X30) sur la pile.

La seconde instruction, `ADD X29, SP, 0` crée la pile. Elle écrit simplement la valeur de SP dans X29.

Ensuite nous voyons la paire d'instructions habituelle ADRP/ADD, qui crée le pointeur sur la chaîne. *lo12* signifie les 12 bits de poids faible, i.e., le linker va écrire les 12 bits de poids faible de l'adresse LC1 dans l'opcode de l'instruction ADD. 1, 2 et 3 sont des valeurs *int* 32-bit, dons elles sont chargées dans les parties 32-bit des registres[69]

GCC (Linaro) 4.9 avec optimisation génère le même code.

[69] Changer 1 par 1L en fera une valeur 64-bit qui sera chargée dans un registres 64-bit. En lire plus sur les entiers constants/littéraux: 1, 2.

ARM: 8 arguments entier

Utilisons de nouveau l'exemple avec 9 arguments de la section précédente: 1.11.1 on page 70.

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3,↙
    ↳  4, 5, 6, 7, 8);
        return 0;
};
```

avec optimisation Keil 6/2013 : Mode ARM

```
.text:00000028             main
.text:00000028
.text:00000028             var_18 = -0x18
.text:00000028             var_14 = -0x14
.text:00000028             var_4  = -4
.text:00000028
.text:00000028 04 E0 2D E5  STR    LR, [SP,#var_4]!
.text:0000002C 14 D0 4D E2  SUB    SP, SP, #0x14
.text:00000030 08 30 A0 E3  MOV    R3, #8
.text:00000034 07 20 A0 E3  MOV    R2, #7
.text:00000038 06 10 A0 E3  MOV    R1, #6
.text:0000003C 05 00 A0 E3  MOV    R0, #5
.text:00000040 04 C0 8D E2  ADD    R12, SP, #0x18+var_14
.text:00000044 0F 00 8C E8  STMIA  R12, {R0-R3}
.text:00000048 04 00 A0 E3  MOV    R0, #4
.text:0000004C 00 00 8D E5  STR    R0, [SP,#0x18+var_18]
.text:00000050 03 30 A0 E3  MOV    R3, #3
.text:00000054 02 20 A0 E3  MOV    R2, #2
.text:00000058 01 10 A0 E3  MOV    R1, #1
.text:0000005C 6E 0F 8F E2  ADR    R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d;
    d=%d; e=%d; f=%d; g=%"...
.text:00000060 BC 18 00 EB  BL     __2printf
.text:00000064 14 D0 8D E2  ADD    SP, SP, #0x14
.text:00000068 04 F0 9D E4  LDR    PC, [SP+4+var_4],#4
```

Ce code peut être divisé en plusieurs parties:

- Prologue de la fonction:

  La toute première instruction `STR LR, [SP,#var_4]!` sauve LR sur la pile, car nous allons utiliser ce registre pour l'appel à `printf()`. Le point d'exclamation à la fin indique un *pré-index*.

  Cela signifie que SP est d'abord décrémenté de 4, et qu'ensuite LR va être sauvé à l'adresse stockée dans SP. C'est similaire à PUSH en x86. Lire aussi à ce propos: 1.39.2 on page 568.

La seconde instruction `SUB SP, SP, #0x14` décrémente SP (le pointeur de pile) afin d'allouer `0x14` (20) octets sur la pile. En effet, nous devons passer 5 valeurs de 32-bit par la pile à la fonction `printf()`, et chacune occupe 4 octets, ce qui fait exactement $5 * 4 = 20$. Les 4 autres valeurs de 32-bit sont passées par les registres.

- Passer 5, 6, 7 et 8 par la pile: ils sont stockés dans les registres `R0`, `R1`, `R2` et `R3` respectivement.
  Ensuite, l'instruction `ADD R12, SP, #0x18+var_14` écrit l'adresse de la pile où ces 4 variables doivent être stockées dans le registre `R12`. *var_14* est une macro d'assemblage, égal à -0x14, créée par IDA pour afficher commodément le code accédant à la pile. Les macros *var_?* générée par IDA reflètent les variables locales dans la pile.

  Donc, `SP+4` doit être stocké dans le registre `R12`.
  L'instruction suivante `STMIA R12, R0-R3` écrit le contenu des registres `R0-R3` dans la mémoire pointée par `R12`. `STMIA` est l'abréviation de *Store Multiple Increment After* (stocker plusieurs incrémenter après). *Increment After* signifie que `R12` doit être incrémenté de 4 après l'écriture de chaque valeur d'un registre.

- Passer 4 par la pile: 4 est stocké dans `R0` et ensuite, cette valeur, avec l'aide de
  l'instruction `STR R0, [SP,#0x18+var_18]` est sauvée dans la pile. *var_18* est -0x18, donc l'offset est 0, donc la valeur du registre `R0` (4) est écrite à l'adresse écrite dans SP.

- Passer 1, 2 et 3 par des registres: Les valeurs des 3 premiers nombres (a,b,c) (respectivement 1, 2, 3) sont passées par les registres `R1`, `R2` et `R3` juste avant l'appel de `printf()`.

- appel de `printf()`

- Épilogue de fonction:

  L'instruction `ADD SP, SP, #0x14` restaure le pointeur SP à sa valeur précédente, nettoyant ainsi la pile. Bien sûr, ce qui a été stocké sur la pile y reste, mais sera récrit lors de l'exécution ultérieure de fonctions.

  L'instruction `LDR PC, [SP+4+var_4],#4` charge la valeur sauvée de LR depuis la pile dans le registre PC, provoquant ainsi la sortie de la fonction. Il n'y a pas de point d'exclamation—effectivement, PC est d'abord chargé depuis l'adresse stockées dans SP ($4 + var_4 = 4 + (-4) = 0$), donc cette instruction est analogue à INSLDR PC, [SP],#4), et ensuite SP est incrémenté de 4. Il s'agit de *post-index*[70]. Pourquoi est-ce qu'IDA affiche l'instruction comme ça? Parce qu'il veut illustrer la disposition de la pile et le fait que `var_4` est alloué pour sauver la valeur de LR dans la pile locale. Cette instruction est quelque peu similaire à `POP PC` en x86[71].

[70]Lire à ce propos: 1.39.2 on page 568.
[71]Il est impossible de définir la valeur de `IP/EIP/RIP` en utilisant `POP` en x86, mais de toutes façons, vous avez le droit de faire l'analogie.

avec optimisation Keil 6/2013 : Mode Thumb

```
.text:0000001C             printf_main2
.text:0000001C
.text:0000001C             var_18 = -0x18
.text:0000001C             var_14 = -0x14
.text:0000001C             var_8  = -8
.text:0000001C
.text:0000001C 00 B5        PUSH    {LR}
.text:0000001E 08 23        MOVS    R3, #8
.text:00000020 85 B0        SUB     SP, SP, #0x14
.text:00000022 04 93        STR     R3, [SP,#0x18+var_8]
.text:00000024 07 22        MOVS    R2, #7
.text:00000026 06 21        MOVS    R1, #6
.text:00000028 05 20        MOVS    R0, #5
.text:0000002A 01 AB        ADD     R3, SP, #0x18+var_14
.text:0000002C 07 C3        STMIA   R3!, {R0-R2}
.text:0000002E 04 20        MOVS    R0, #4
.text:00000030 00 90        STR     R0, [SP,#0x18+var_18]
.text:00000032 03 23        MOVS    R3, #3
.text:00000034 02 22        MOVS    R2, #2
.text:00000036 01 21        MOVS    R1, #1
.text:00000038 A0 A0        ADR     R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d;
    d=%d; e=%d; f=%d; g=%"...
.text:0000003A 06 F0 D9 F8  BL      __2printf
.text:0000003E
.text:0000003E             loc_3E   ; CODE XREF: example13_f+16
.text:0000003E 05 B0        ADD     SP, SP, #0x14
.text:00000040 00 BD        POP     {PC}
```

La sortie est presque comme dans les exemples précédents. Toutefois, c'est du code Thumb et les valeurs sont arrangées différemment dans la pile: 8 vient en premier, puis 5, 6, 7 et 4 vient en troisième.

avec optimisation Xcode 4.6.3 (LLVM) : Mode ARM

```
__text:0000290C             _printf_main2
__text:0000290C
__text:0000290C             var_1C = -0x1C
__text:0000290C             var_C  = -0xC
__text:0000290C
__text:0000290C 80 40 2D E9   STMFD  SP!, {R7,LR}
__text:00002910 0D 70 A0 E1   MOV    R7, SP
__text:00002914 14 D0 4D E2   SUB    SP, SP, #0x14
__text:00002918 70 05 01 E3   MOV    R0, #0x1570
__text:0000291C 07 C0 A0 E3   MOV    R12, #7
__text:00002920 00 00 40 E3   MOVT   R0, #0
__text:00002924 04 20 A0 E3   MOV    R2, #4
__text:00002928 00 00 8F E0   ADD    R0, PC, R0
__text:0000292C 06 30 A0 E3   MOV    R3, #6
__text:00002930 05 10 A0 E3   MOV    R1, #5
```

```
__text:00002934 00 20 8D E5   STR    R2, [SP,#0x1C+var_1C]
__text:00002938 0A 10 8D E9   STMFA  SP, {R1,R3,R12}
__text:0000293C 08 90 A0 E3   MOV    R9, #8
__text:00002940 01 10 A0 E3   MOV    R1, #1
__text:00002944 02 20 A0 E3   MOV    R2, #2
__text:00002948 03 30 A0 E3   MOV    R3, #3
__text:0000294C 10 90 8D E5   STR    R9, [SP,#0x1C+var_C]
__text:00002950 A4 05 00 EB   BL     _printf
__text:00002954 07 D0 A0 E1   MOV    SP, R7
__text:00002958 80 80 BD E8   LDMFD  SP!, {R7,PC}
```

Presque la même chose que ce que nous avons déjà vu, avec l'exception de l'instruction STMFA (Store Multiple Full Ascending), qui est un synonyme de l'instruction STMIB (Store Multiple Increment Before). Cette instruction incrémente la valeur du registre SP et écrit seulement après la valeur registre suivant dans la mémoire, plutôt que d'effectuer ces deux actions dans l'ordre inverse.

Une autre chose qui accroche le regard est que les instructions semblent être arrangées de manière aléatoire. Par exemple, la valeur dans le registre R0 est manipulée en trois endroits, aux adresses 0x2918, 0x2920 et 0x2928, alors qu'il serait possible de le faire en un seul endroit.

Toutefois, le compilateur qui optimise doit avoir ses propres raisons d'ordonner les instructions pour avoir une plus grande efficacité à l'exécution.

D'habitude, le processeur essaye d'exécuter simultanément des instructions situées côte à côte.
Par exemple, des instructions comme MOVT R0, #0 et ADD R0, PC, R0 ne peuvent pas être exécutées simultanément puisqu'elles modifient toutes deux le registre R0. D'un autre côté, les instructions MOVT R0, #0 et MOV R2, #4 peuvent être exécutées simultanément puisque leurs effets n'interfèrent pas l'un avec l'autre lors de leurs exécution. Probablement que le compilateur essaye de générer du code arrangé de cette façon (lorsque c'est possible).

avec optimisation Xcode 4.6.3 (LLVM) : Mode Thumb-2

```
__text:00002BA0               _printf_main2
__text:00002BA0
__text:00002BA0               var_1C = -0x1C
__text:00002BA0               var_18 = -0x18
__text:00002BA0               var_C  = -0xC
__text:00002BA0
__text:00002BA0 80 B5          PUSH     {R7,LR}
__text:00002BA2 6F 46          MOV      R7, SP
__text:00002BA4 85 B0          SUB      SP, SP, #0x14
__text:00002BA6 41 F2 D8 20    MOVW     R0, #0x12D8
__text:00002BAA 4F F0 07 0C    MOV.W    R12, #7
__text:00002BAE C0 F2 00 00    MOVT.W   R0, #0
__text:00002BB2 04 22          MOVS     R2, #4
__text:00002BB4 78 44          ADD      R0, PC  ; char *
__text:00002BB6 06 23          MOVS     R3, #6
```

```
__text:00002BB8 05 21           MOVS     R1, #5
__text:00002BBA 0D F1 04 0E     ADD.W    LR, SP, #0x1C+var_18
__text:00002BBE 00 92           STR      R2, [SP,#0x1C+var_1C]
__text:00002BC0 4F F0 08 09     MOV.W    R9, #8
__text:00002BC4 8E E8 0A 10     STMIA.W  LR, {R1,R3,R12}
__text:00002BC8 01 21           MOVS     R1, #1
__text:00002BCA 02 22           MOVS     R2, #2
__text:00002BCC 03 23           MOVS     R3, #3
__text:00002BCE CD F8 10 90     STR.W    R9, [SP,#0x1C+var_C]
__text:00002BD2 01 F0 0A EA     BLX      _printf
__text:00002BD6 05 B0           ADD      SP, SP, #0x14
__text:00002BD8 80 BD           POP      {R7,PC}
```

La sortie est presque la même que dans l'exemple précédent, avec l'exception que des instructions Thumb/Thumb-2 sont utilisées à la place.

ARM64

GCC (Linaro) 4.9 sans optimisation

Listing 1.56 : GCC (Linaro) 4.9 sans optimisation

```
.LC2:
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
f3:
; Réserver plus d'espace dans la pile:
        sub     sp, sp, #32
; sauver FP et LR sur la pile:
        stp     x29, x30, [sp,16]
; définir la pile (FP=SP+16):
        add     x29, sp, 16
        adrp    x0, .LC2 ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
        add     x0, x0, :lo12:.LC2
        mov     w1, 8           ; 9ème argument
        str     w1, [sp]        ; stocker le 9ème argument dans la pile
        mov     w1, 1
        mov     w2, 2
        mov     w3, 3
        mov     w4, 4
        mov     w5, 5
        mov     w6, 6
        mov     w7, 7
        bl      printf
        sub     sp, x29, #16
; restaurer FP et LR
        ldp     x29, x30, [sp,16]
        add     sp, sp, 32
        ret
```

Les 8 premiers arguments sont passés dans des registres X- ou W-: [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][72]. Un pointeur de chaîne nécessite un registre 64-bit, donc il est passé dans X0. Toutes les autres valeurs ont un type *int* 32-bit, donc elles sont stockées dans la partie 32-bit des registres (W-). Le 9ème argument (8) est passé par la pile. En effet: il n'est pas possible de passer un grand nombre d'arguments par les registres, car le nombre de registres est limité.

GCC (Linaro) 4.9 avec optimisation génère le même code.

1.11.3 MIPS

3 arguments entier

GCC 4.4.5 avec optimisation

La différence principale avec l'exemple «Hello, world! » est que dans ce cas, `printf()` est appelée à la place de `puts()` et 3 arguments de plus sont passés à travers les registres $5…$7 (ou $A0…$A2). C'est pourquoi ces registres sont préfixés avec A-, ceci sous-entend qu'ils sont utilisés pour le passage des arguments aux fonctions.

Listing 1.57 : GCC 4.4.5 avec optimisation (résultat en sortie de l'assembleur)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d\000"
main:
; prologue de la fonction:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-32
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,28($sp)
; charger l'adresse de printf():
        lw      $25,%call16(printf)($28)
; charger l'adresse de la chaîne de texte et mettre le 1er argument de
    printf():
        lui     $4,%hi($LC0)
        addiu   $4,$4,%lo($LC0)
; mettre le 2nd argument de printf():
        li      $5,1                    # 0x1
; mettre le 3ème argument de printf():
        li      $6,2                    # 0x2
; appeler printf():
        jalr    $25
; mettre le 4ème argument de printf() (slot de délai branchement):
        li      $7,3                    # 0x3

; épilogue de la fonction:
        lw      $31,28($sp)
; mettre la valeur de retour à 0:
        move    $2,$0
; retourner
```

[72] Aussi disponible en http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

```
        j       $31
        addiu   $sp,$sp,32 ; slot de délai de branchement
```

Listing 1.58 : GCC 4.4.5 avec optimisation (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; prologue de la fonction:
.text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                 addiu   $sp, -0x20
.text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C                 sw      $ra, 0x20+var_4($sp)
.text:00000010                 sw      $gp, 0x20+var_10($sp)
; charger l'adresse de printf():
.text:00000014                 lw      $t9, (printf & 0xFFFF)($gp)
; charger l'adresse de la chaîne de texte et mettre le 1er argument de
    printf():
.text:00000018                 la      $a0, $LC0        # "a=%d; b=%d; c=%d"
; mettre le 2nd argument de printf():
.text:00000020                 li      $a1, 1
; mettre le 3ème argument de printf():
.text:00000024                 li      $a2, 2
; appeler printf():
.text:00000028                 jalr    $t9
; mettre le 4ème argument de printf(): (slot de délai de branchement)
.text:0000002C                 li      $a3, 3
; épilogue de la fonction:
.text:00000030                 lw      $ra, 0x20+var_4($sp)
; mettre la valeur de retour à 0:
.text:00000034                 move    $v0, $zero
; retourner
.text:00000038                 jr      $ra
.text:0000003C                 addiu   $sp, 0x20 ; slot de délai de
    branchement
```

IDA a agrégé la paire d'instructions `LUI` et `ADDIU` en une pseudo instruction `LA`. C'est pourquoi il n'y a pas d'instruction à l'adresse 0x1C: car `LA` *occupe* 8 octets.

GCC 4.4.5 sans optimisation

GCC sans optimisation est plus verbeux:

Listing 1.59 : GCC 4.4.5 sans optimisation (résultat en sortie de l'assembleur)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d\000"
main:
; prologue de la fonction:
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
```

```
        sw      $fp,24($sp)
        move    $fp,$sp
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
; charger l'adresse de la chaîne de texte:
        lui     $2,%hi($LC0)
        addiu   $2,$2,%lo($LC0)
; mettre le 1er argument de printf():
        move    $4,$2
; mettre le 2nd argument de printf():
        li      $5,1                    # 0x1
; mettre le 3ème argument de printf():
        li      $6,2                    # 0x2
; mettre le 4ème argument de printf():
        li      $7,3                    # 0x3
; charger l'adresse de printf():
        lw      $2,%call16(printf)($28)
        nop
; appeler printf():
        move    $25,$2
        jalr    $25
        nop

; épilogue de la fonction:
        lw      $28,16($fp)
; mettre la valeur de retour à 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,28($sp)
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
; retourner
        j       $31
        nop
```

Listing 1.60 : GCC 4.4.5 sans optimisation (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; prologue de la fonction:
.text:00000000                 addiu   $sp, -0x20
.text:00000004                 sw      $ra, 0x20+var_4($sp)
.text:00000008                 sw      $fp, 0x20+var_8($sp)
.text:0000000C                 move    $fp, $sp
.text:00000010                 la      $gp, __gnu_local_gp
.text:00000018                 sw      $gp, 0x20+var_10($sp)
; charge l'adresse de la chaîne de texte:
.text:0000001C                 la      $v0, aADBDCD     # "a=%d; b=%d; c=%d"
; mettre le 1er argument de printf():
.text:00000024                 move    $a0, $v0
```

```
; mettre le 2nd argument de printf():
.text:00000028                 li      $a1, 1
; mettre le 3ème argument de printf():
.text:0000002C                 li      $a2, 2
; mettre le 4ème argument de printf():
.text:00000030                 li      $a3, 3
; charger l'adresse de printf():
.text:00000034                 lw      $v0, (printf & 0xFFFF)($gp)
.text:00000038                 or      $at, $zero
; appeler printf():
.text:0000003C                 move    $t9, $v0
.text:00000040                 jalr    $t9
.text:00000044                 or      $at, $zero ; NOP
; épilogue de la fonction:
.text:00000048                 lw      $gp, 0x20+var_10($fp)
; mettre la valeur de retour à 0:
.text:0000004C                 move    $v0, $zero
.text:00000050                 move    $sp, $fp
.text:00000054                 lw      $ra, 0x20+var_4($sp)
.text:00000058                 lw      $fp, 0x20+var_8($sp)
.text:0000005C                 addiu   $sp, 0x20
; retourner
.text:00000060                 jr      $ra
.text:00000064                 or      $at, $zero ; NOP
```

8 arguments entier

Utilisons encore l'exemple de la section précédente avec 9 arguments: 1.11.1 on page 70.

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3,↙
    ↘ 4, 5, 6, 7, 8);
        return 0;
};
```

GCC 4.4.5 avec optimisation

Seul les 4 premiers arguments sont passés dans les registres $A0 …$A3, les autres sont passés par la pile.

C'est la convention d'appel O32 (qui est la plus commune dans le monde MIPS). D'autres conventions d'appel, ou du code assembleur écrit à la main, peuvent utiliser les registres à d'autres fins.

SW est l'abréviation de «Store Word» (depuis un registre vers la mémoire). En MIPS, il manque une instructions pour stocker une valeur dans la mémoire, donc une paire d'instruction doit être utilisée à la place (LI/SW).

Listing 1.61 : GCC 4.4.5 avec optimisation (résultat en sortie de l'assembleur)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; prologue de la fonction:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-56
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,52($sp)
; passer le 5ème argument dans la pile:
        li      $2,4                    # 0x4
        sw      $2,16($sp)
; passer le 6ème argument dans la pile:
        li      $2,5                    # 0x5
        sw      $2,20($sp)
; passer le 7ème argument dans la pile:
        li      $2,6                    # 0x6
        sw      $2,24($sp)
; passer le 8ème argument dans la pile:
        li      $2,7                    # 0x7
        lw      $25,%call16(printf)($28)
        sw      $2,28($sp)
; passer le 1er argument dans $a0:
        lui     $4,%hi($LC0)
; passer le 9ème argument dans la pile:
        li      $2,8                    # 0x8
        sw      $2,32($sp)
        addiu   $4,$4,%lo($LC0)
; passer le 2nd argument dans $a1:
        li      $5,1                    # 0x1
; passer le 3ème argument dans $a2:
        li      $6,2                    # 0x2
; appeler printf():
        jalr    $25
; passer le 4ème argument dans $a3 (slot de délai de branchement):
        li      $7,3                    # 0x3

; épilogue de la fonction:
        lw      $31,52($sp)
; mettre la valeur de retour à 0:
        move    $2,$0
; retourner
        j       $31
        addiu   $sp,$sp,56 ; slot de délai de branchement
```

Listing 1.62 : GCC 4.4.5 avec optimisation (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
.text:00000000 var_1C          = -0x1C
```

```
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; prologue de la fonction:
.text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                 addiu   $sp, -0x38
.text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C                 sw      $ra, 0x38+var_4($sp)
.text:00000010                 sw      $gp, 0x38+var_10($sp)
; passer le 5ème argument dans la pile:
.text:00000014                 li      $v0, 4
.text:00000018                 sw      $v0, 0x38+var_28($sp)
; passer le 6ème argument dans la pile:
.text:0000001C                 li      $v0, 5
.text:00000020                 sw      $v0, 0x38+var_24($sp)
; passer le 7ème argument dans la pile:
.text:00000024                 li      $v0, 6
.text:00000028                 sw      $v0, 0x38+var_20($sp)
; passer le 8ème argument dans la pile:
.text:0000002C                 li      $v0, 7
.text:00000030                 lw      $t9, (printf & 0xFFFF)($gp)
.text:00000034                 sw      $v0, 0x38+var_1C($sp)
; préparer le 1er argument dans $a0:
.text:00000038                 lui     $a0, ($LC0 >> 16)  # "a=%d; b=%d;
    c=%d; d=%d; e=%d; f=%d; g=%"...
; passer le 9ème argument dans la pile:
.text:0000003C                 li      $v0, 8
.text:00000040                 sw      $v0, 0x38+var_18($sp)
; passer le 1er argument in $a0:
.text:00000044                 la      $a0, ($LC0 & 0xFFFF)  # "a=%d; b=%d;
    c=%d; d=%d; e=%d; f=%d; g=%"...
; passer le 2nd argument dans $a1:
.text:00000048                 li      $a1, 1
; passer le 3ème argument dans $a2:
.text:0000004C                 li      $a2, 2
; appeler printf():
.text:00000050                 jalr    $t9
; passer le 4ème argument dans $a3 (slot de délai de branchement):
.text:00000054                 li      $a3, 3
; épilogue de la fonction:
.text:00000058                 lw      $ra, 0x38+var_4($sp)
; mettre la valeur de retour à 0:
.text:0000005C                 move    $v0, $zero
; retourner
.text:00000060                 jr      $ra
.text:00000064                 addiu   $sp, 0x38 ; slot de délai de
    branchement
```

GCC 4.4.5 sans optimisation

GCC sans optimisation est plus verbeux:

Listing 1.63 : sans optimisation GCC 4.4.5 (résultat en sortie de l'assembleur)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; prologue de la fonction:
        addiu   $sp,$sp,-56
        sw      $31,52($sp)
        sw      $fp,48($sp)
        move    $fp,$sp
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
        lui     $2,%hi($LC0)
        addiu   $2,$2,%lo($LC0)
; passer le 5ème argument dans la pile:
        li      $3,4                    # 0x4
        sw      $3,16($sp)
; passer le 6ème argument dans la pile:
        li      $3,5                    # 0x5
        sw      $3,20($sp)
; passer le 7ème argument dans la pile:
        li      $3,6                    # 0x6
        sw      $3,24($sp)
; passer le 8ème argument dans la pile:
        li      $3,7                    # 0x7
        sw      $3,28($sp)
; passer le 9ème argument dans la pile:
        li      $3,8                    # 0x8
        sw      $3,32($sp)
; passer le 1er argument dans $a0:
        move    $4,$2
; passer le 2nd argument dans $a1:
        li      $5,1                    # 0x1
; passer le 3ème argument dans $a2:
        li      $6,2                    # 0x2
; passer le 4ème argument dans $a3:
        li      $7,3                    # 0x3
; appeler printf():
        lw      $2,%call16(printf)($28)
        nop
        move    $25,$2
        jalr    $25
        nop
; épilogue de la fonction:
        lw      $28,40($fp)
; mettre la valeur de retour à 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,52($sp)
        lw      $fp,48($sp)
        addiu   $sp,$sp,56
; retourner
        j       $31
        nop
```

Listing 1.64 : sans optimisation GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
.text:00000000 var_1C          = -0x1C
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; prologue de la fonction:
.text:00000000                 addiu   $sp, -0x38
.text:00000004                 sw      $ra, 0x38+var_4($sp)
.text:00000008                 sw      $fp, 0x38+var_8($sp)
.text:0000000C                 move    $fp, $sp
.text:00000010                 la      $gp, __gnu_local_gp
.text:00000018                 sw      $gp, 0x38+var_10($sp)
.text:0000001C                 la      $v0, aADBDCDDDEDFDGD  # "a=%d; b=%d;
    c=%d; d=%d; e=%d; f=%d; g=%"...
; passer le 5ème argument dans la pile:
.text:00000024                 li      $v1, 4
.text:00000028                 sw      $v1, 0x38+var_28($sp)
; passer le 6ème argument dans la pile:
.text:0000002C                 li      $v1, 5
.text:00000030                 sw      $v1, 0x38+var_24($sp)
; passer le 7ème argument dans la pile:
.text:00000034                 li      $v1, 6
.text:00000038                 sw      $v1, 0x38+var_20($sp)
; passer le 8ème argument dans la pile:
.text:0000003C                 li      $v1, 7
.text:00000040                 sw      $v1, 0x38+var_1C($sp)
; passer le 9ème argument dans la pile:
.text:00000044                 li      $v1, 8
.text:00000048                 sw      $v1, 0x38+var_18($sp)
; passer le 1er argument dans $a0:
.text:0000004C                 move    $a0, $v0
; passer le 2nd argument dans $a1:
.text:00000050                 li      $a1, 1
; passer le 3ème argument dans $a2:
.text:00000054                 li      $a2, 2
; passer le 4ème argument dans $a3:
.text:00000058                 li      $a3, 3
; appeler printf():
.text:0000005C                 lw      $v0, (printf & 0xFFFF)($gp)
.text:00000060                 or      $at, $zero
.text:00000064                 move    $t9, $v0
.text:00000068                 jalr    $t9
.text:0000006C                 or      $at, $zero ; NOP
; épilogue de la fonction:
.text:00000070                 lw      $gp, 0x38+var_10($fp)
; mettre la valeur de retour à 0:
```

```
.text:00000074                 move    $v0, $zero
.text:00000078                 move    $sp, $fp
.text:0000007C                 lw      $ra, 0x38+var_4($sp)
.text:00000080                 lw      $fp, 0x38+var_8($sp)
.text:00000084                 addiu   $sp, 0x38
; retourner
.text:00000088                 jr      $ra
.text:0000008C                 or      $at, $zero ; NOP
```

1.11.4 Conclusion

Voici un schéma grossier de l'appel de la fonction:

Listing 1.65 : x86

```
...
PUSH 3rd argument
PUSH 2nd argument
PUSH 1st argument
CALL fonction
; modifier le pointeur de pile (si besoin)
```

Listing 1.66 : x64 (MSVC)

```
MOV RCX, 1st argument
MOV RDX, 2nd argument
MOV R8, 3rd argument
MOV R9, 4th argument
...
PUSH 5ème, 6ème argument, etc. (si besoin)
CALL fonction
; modifier le pointeur de pile (si besoin)
```

Listing 1.67 : x64 (GCC)

```
MOV RDI, 1st argument
MOV RSI, 2nd argument
MOV RDX, 3rd argument
MOV RCX, 4th argument
MOV R8, 5th argument
MOV R9, 6th argument
...
PUSH 7ème, 8ème argument, etc. (si besoin)
CALL fonction
; modifier le pointeur de pile (si besoin)
```

Listing 1.68 : ARM

```
MOV R0, 1st argument
MOV R1, 2nd argument
MOV R2, 3rd argument
MOV R3, 4th argument
; passer le 5ème, 6ème argument, etc., dans la pile (si besoin)
```

```
BL fonction
; modifier le pointeur de pile (si besoin)
```

Listing 1.69 : ARM64

```
MOV X0, 1st argument
MOV X1, 2nd argument
MOV X2, 3rd argument
MOV X3, 4th argument
MOV X4, 5th argument
MOV X5, 6th argument
MOV X6, 7th argument
MOV X7, 8th argument
; passer le 9ème, 10ème argument, etc., dans la pile (si besoin)
BL fonction
; modifier le pointeur de pile (si besoin)
```

Listing 1.70 : MIPS (O32 calling convention)

```
LI $4, 1st argument ; AKA $A0
LI $5, 2nd argument ; AKA $A1
LI $6, 3rd argument ; AKA $A2
LI $7, 4th argument ; AKA $A3
; passer le 5ème, 6ème argument, etc., dans la pile (si besoin)
LW temp_reg, adresse de la fonction
JALR temp_reg
```

1.11.5 À propos

À propos, cette différence dans le passage des arguments entre x86, x64, fastcall, ARM et MIPS est une bonne illustration du fait que le CPU est inconscient de comment les arguments sont passés aux fonctions. Il est aussi possible de créer un compilateur capable de passer les arguments via une structure spéciale sans utiliser du tout la pile.

Les registres MIPS $A0 …$A3 sont appelés comme ceci par commodité (c'est dans la convention d'appel O32). Les programmeurs peuvent utiliser n'importe quel autre registre, (bon, peut-être à l'exception de $ZERO) pour passer des données ou n'importe quelle autre convention d'appel.

Le CPU n'est pas au courant de quoi que ce soit des conventions d'appel.

Nous pouvons aussi nous rappeler comment les débutants en langage d'assemblage passent les arguments aux autres fonctions: généralement par les registres, sans ordre explicite, ou même par des variables globales. Bien sûr, cela fonctionne.

1.12 scanf()

Maintenant utilisons la fonction `scanf()`.

1.12.1 Exemple simple

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

Il n'est pas astucieux d'utiliser `scanf()` pour les interactions utilisateurs de nos jours. Mais nous pouvons, toutefois, illustrer le passage d'un pointeur sur une variable de type *int*.

À propos des pointeurs

Les pointeurs sont l'un des concepts fondamentaux de l'informatique. Souvent, passer un gros tableau, structure ou objet comme argument à une autre fonction est trop coûteux, tandis que passer leur adresse l'est très peu. Par exemple, si vous voulez afficher une chaîne de texte sur la console, il est plus facile de passer son adresse au noyau de l'OS.

En plus, si la fonction appelée doit modifier quelque chose dans un gros tableau ou structure reçu comme paramètre et renvoyer le tout, la situation est proche de l'absurde. Donc, la chose la plus simple est de passer l'adresse du tableau ou de la structure à la fonction appelée, et de la laisser changer ce qui doit l'être.

Un pointeur en C/C++—est simplement l'adresse d'un emplacement mémoire quelconque.

En x86, l'adresse est représentée par un nombre de 32-bit (i.e., il occupe 4 octets), tandis qu'en x86-64 c'est un nombre de 64-bit (occupant 8 octets). À propos, c'est la cause de l'indignation de certaines personnes concernant le changement vers x86-64—tous les pointeurs en architecture x64 ayant besoin de deux fois plus de place, incluant la mémoire cache, qui est de la mémoire "coûteuse".

Il est possible de travailler seulement avec des pointeurs non typés, moyennant quelques efforts; e.g. la fonction C standard `memcpy()`, qui copie un bloc de mémoire d'un endroit à un autre, prend 2 pointeurs de type `void*` comme arguments, puisqu'il est impossible de prévoir le type de données qu'il faudra copier. Les types de données ne sont pas importants, seule la taille du bloc compte.

Les pointeurs sont aussi couramment utilisés lorsqu'une fonction doit renvoyer plus d'une valeur (nous reviendrons là-dessus plus tard (3.23 on page 773)).

La fonction *scanf()*—en est une telle.

Hormis le fait que la fonction doit indiquer combien de valeurs ont été lues avec succès, elle doit aussi renvoyer toutes ces valeurs.

En C/C++ le type du pointeur est seulement nécessaire pour la vérification de type lors de la compilation.

Il n'y a aucune information du tout sur le type des pointeurs à l'intérieur du code compilé.

x86

MSVC

Voici ce que l'on obtient après avoir compilé avec MSVC 2010:

```
CONST    SEGMENT
$SG3831    DB    'Enter X:', 0aH, 00H
$SG3832    DB    '%d', 00H
$SG3833    DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Options de compilation de la fonction: /Odtp
_TEXT    SEGMENT
_x$ = -4                        ; size = 4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    push   OFFSET $SG3831 ; 'Enter X:'
    call   _printf
    add    esp, 4
    lea    eax, DWORD PTR _x$[ebp]
    push   eax
    push   OFFSET $SG3832 ; '%d'
    call   _scanf
    add    esp, 8
    mov    ecx, DWORD PTR _x$[ebp]
    push   ecx
    push   OFFSET $SG3833 ; 'You entered %d...'
    call   _printf
    add    esp, 8

    ; retourner 0
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

x est une variable locale.

D'après le standard C/C++ elle ne doit être visible que dans cette fonction et dans aucune autre portée. Traditionnellement, les variables locales sont stockées sur la pile. Il y a probablement d'autres moyens de les allouer, mais en x86, c'est la façon de faire.

Le but de l'instruction suivant le prologue de la fonction, PUSH ECX, n'est pas de sauver l'état de ECX (noter l'absence d'un POP ECX à la fin de la fonction).

En fait, cela alloue 4 octets sur la pile pour stocker la variable x.

x est accédée à l'aide de la macro _x$ (qui vaut -4) et du registre EBP qui pointe sur la structure de pile courante.

Pendant la durée de l'exécution de la fonction, EBP pointe sur la structure locale de pile courante, rendant possible l'accès aux variables locales et aux arguments de la fonction via EBP+offset.

Il est aussi possible d'utiliser ESP dans le même but, bien que ça ne soit pas très commode, car il change fréquemment. La valeur de EBP peut être perçue comme un *état figé* de la valeur de ESP au début de l'exécution de la fonction.

Voici une structure de pile typique dans un environnement 32-bit:

…	…
EBP-8	variable locale #2, marqué dans IDA comme var_8
EBP-4	variable locale #1, marqué dans IDA comme var_4
EBP	valeur sauvée de EBP
EBP+4	adresse de retour
EBP+8	argument#1, marqué dans IDA comme arg_0
EBP+0xC	argument#2, marqué dans IDA comme arg_4
EBP+0x10	argument#3, marqué dans IDA comme arg_8
…	…

La fonction scanf() de notre exemple a deux arguments.

Le premier est un pointeur sur la chaîne contenant %d et le second est l'adresse de la variable x.

Tout d'abord, l'adresse de la variable x est chargée dans le registre EAX par l'instruction
lea eax, DWORD PTR _x$[ebp].

LEA signifie *load effective address* (charger l'adresse effective) et est souvent utilisée pour composer une adresse (.1.6 on page 1333).

Nous pouvons dire que dans ce cas, LEA stocke simplement la somme de la valeur du registre EBP et de la macro _x$ dans le registre EAX.

C'est la même chose que lea eax, [ebp-4].

Donc, 4 est soustrait de la valeur du registre EBP et le résultat est chargé dans le registre EAX. Ensuite, la valeur du registre EAX est poussée sur la pile et scanf() est appelée.

printf() est appelée ensuite avec son premier argument — un pointeur sur la chaîne: You entered %d...\n.

Le second argument est préparé avec: `mov ecx, [ebp-4]`. L'instruction stocke la valeur de la variable x et non son adresse, dans le registre ECX.

Puis, la valeur de ECX est stockée sur la pile et le dernier appel à `printf()` est effectué.

MSVC + OllyDbg

Essayons cet exemple dans OllyDbg. Chargeons-le et appuyons sur F8 (enjamber) jusqu'à ce que nous atteignons notre exécutable au lieu de `ntdll.dll`. Défiler vers le haut jusqu'à ce que `main()` apparaisse.

Cliquer sur la première instruction (PUSH EBP), appuyer sur F2 (*set a breakpoint*), puis F9 (*Run*). Le point d'arrêt sera déclenché lorsque `main()` commencera.

Continuons jusqu'au point où la variable x est calculée:

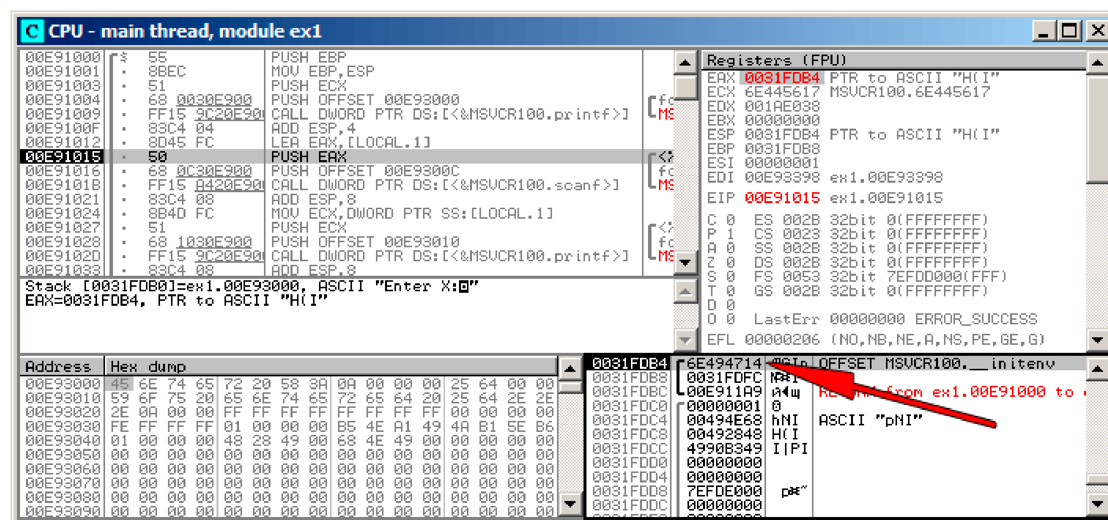


Fig. 1.13: OllyDbg : L'adresse de la variable locale est calculée

Cliquer droit sur EAX dans la fenêtre des registres et choisir «Follow in stack».

Cette adresse va apparaître dans la fenêtre de la pile. La flèche rouge a été ajoutée, pointant la variable dans la pile locale. A ce point, cet espace contient des restes de données (0x6E494714). Maintenant. avec l'aide de l'instruction PUSH, l'adresse de cet élément de pile va être stockée sur la même pile à la position suivante. Appuyons sur F8 jusqu'à la fin de l'exécution de `scanf()`. Pendant l'exécution de `scanf()`, entrons, par exemple, 123, dans la fenêtre de la console:

```
Enter X:
123
```

`scanf()` a déjà fini de s'exécuter:

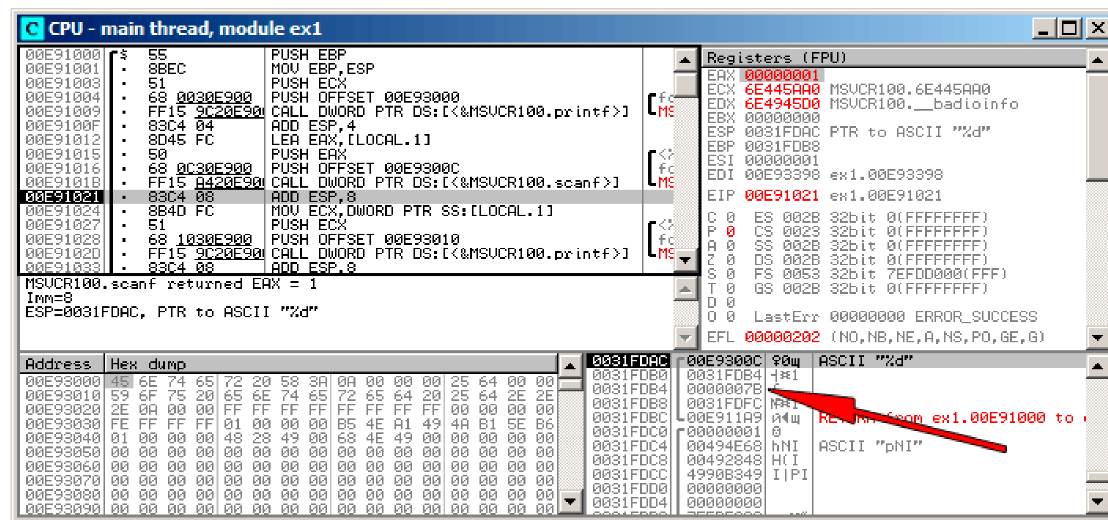


Fig. 1.14: OllyDbg : `scanf()` s'est exécutée

`scanf()` renvoie 1 dans EAX, ce qui indique qu'elle a lu avec succès une valeur. Si nous regardons de nouveau l'élément de la pile correspondant à la variable locale, il contient maintenant `0x7B` (123).

Plus tard, cette valeur est copiée de la pile vers le registre ECX et passée à `printf()` :

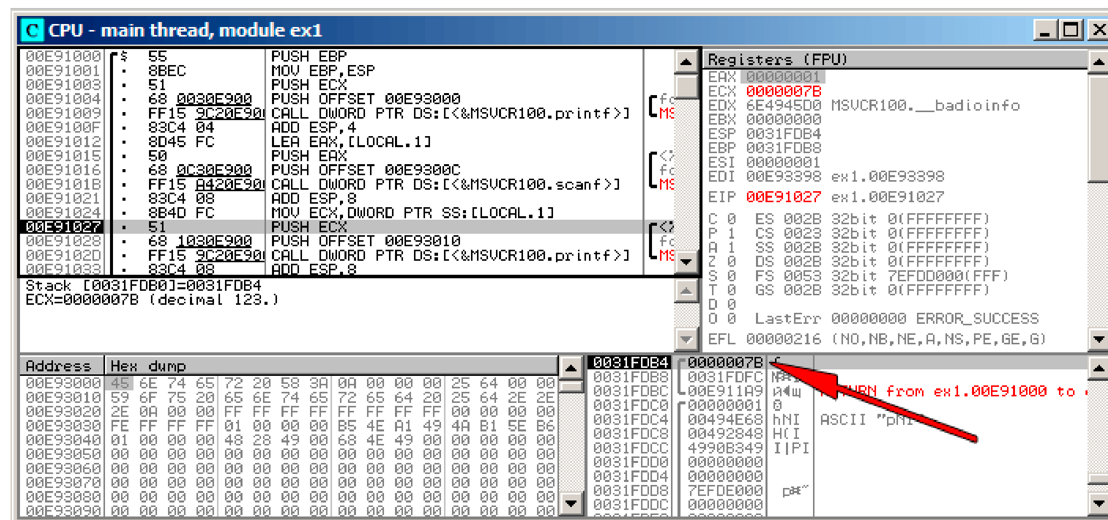


Fig. 1.15: OllyDbg : préparation de la valeur pour la passer à `printf()`

GCC

Compilons ce code avec GCC 4.4.1 sous Linux:

```
main            proc near

var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_20], offset aEnterX ; "Enter X:"
                call    _puts
                mov     eax, offset aD  ; "%d"
                lea     edx, [esp+20h+var_4]
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    ___isoc99_scanf
                mov     edx, [esp+20h+var_4]
                mov     eax, offset aYouEnteredD___ ; "You entered %d...\n"
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    _printf
                mov     eax, 0
                leave
```

```
                retn
main            endp
```

GCC a remplacé l'appel à `printf()` avec un appel à `puts()`. La raison de cela a été expliquée dans (1.5.3 on page 29).

Comme dans l'exemple avec MSVC—les arguments sont placés dans la pile avec l'instruction `MOV`.

À propos

Ce simple exemple est la démonstration du fait que le compilateur traduit une liste d'expression en bloc-C/C++ en une liste séquentielle d'instructions. Il n'y a rien entre les expressions en C/C++, et le résultat en code machine, il n'y a rien entre le déroulement du flux de contrôle d'une expression à la suivante.

x64

Le schéma est ici similaire, avec la différence que les registres, plutôt que la pile, sont utilisés pour le passage des arguments.

MSVC

Listing 1.71 : MSVC 2012 x64

```
_DATA   SEGMENT
$SG1289 DB      'Enter X:', 0aH, 00H
$SG1291 DB      '%d', 00H
$SG1292 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN3:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1291 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
        call    printf

        ; retourner 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
```

```
_TEXT   ENDS
```

GCC

Listing 1.72 : GCC 4.4.6 x64 avec optimisation

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"

main:
        sub     rsp, 24
        mov     edi, OFFSET FLAT:.LC0 ; "Enter X:"
        call    puts
        lea     rsi, [rsp+12]
        mov     edi, OFFSET FLAT:.LC1 ; "%d"
        xor     eax, eax
        call    __isoc99_scanf
        mov     esi, DWORD PTR [rsp+12]
        mov     edi, OFFSET FLAT:.LC2 ; "You entered %d...\n"
        xor     eax, eax
        call    printf

        ; retourner 0
        xor     eax, eax
        add     rsp, 24
        ret
```

ARM

avec optimisation Keil 6/2013 (Mode Thumb)

```
.text:00000042             scanf_main
.text:00000042
.text:00000042             var_8          = -8
.text:00000042
.text:00000042 08 B5        PUSH    {R3,LR}
.text:00000044 A9 A0        ADR     R0, aEnterX  ; "Enter X:\n"
.text:00000046 06 F0 D3 F8  BL      __2printf
.text:0000004A 69 46        MOV     R1, SP
.text:0000004C AA A0        ADR     R0, aD  ; "%d"
.text:0000004E 06 F0 CD F8  BL      __0scanf
.text:00000052 00 99        LDR     R1, [SP,#8+var_8]
.text:00000054 A9 A0        ADR     R0, aYouEnteredD___ ; "You entered
    %d...\n"
.text:00000056 06 F0 CB F8  BL      __2printf
.text:0000005A 00 20        MOVS    R0, #0
```

```
.text:0000005C 08 BD          POP     {R3,PC}
```

Afin que `scanf()` puisse lire l'item, elle a besoin d'un paramètre—un pointeur sur un *int*. Le type *int* est 32-bit, donc nous avons besoin de 4 octets pour le stocker quelque part en mémoire, et il tient exactement dans un registre 32-bit. De l'espace pour la variable locale x est allouée sur la pile et IDA l'a nommée *var_8*. Il n'est toutefois pas nécessaire de définir cette macro, puisque le SP (pointeur de pile) pointe déjà sur cet espace et peut être utilisé directement.

Donc, la valeur de SP est copiée dans la registre R1 et, avec la chaîne de format, passée à `scanf()`.

Les instructions PUSH/POP se comportent différemment en ARM et en x86 (c'est l'inverse) Il y a des sysnonymes aux instructions STM/STMDB/LDM/LDMIA. Et l'instruction PUSH écrit d'abord une valeur sur la pile, *et ensuite* soustrait 4 de SP. De ce fait, après PUSH, SP pointe sur de l'espace inutilisé sur la pile. Il est utilisé par `scanf()`, et après par `printf()`.

LDMIA signifie *Load Multiple Registers Increment address After each transfer* (charge plusieurs registres incrémente l'adresse après chaque transfert). STMDB signifie *Store Multiple Registers Decrement address Before each transfer* (socke plusieurs registres décrémente l'adresse avant chaque transfert).

Plus tard, avec l'aide de l'instruction LDR, cette valeur est copiée depuis la pile vers le registre R1 afin de la passer à `printf()`.

ARM64

Listing 1.73 : GCC 4.9.1 ARM64 sans optimisation

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"
scanf_main:
; soustraire 32 de SP, puis sauver FP et LR dans la structure de pile:
        stp     x29, x30, [sp, -32]!
; utiliser la partie de pile (FP=SP)
        add     x29, sp, 0
; charger le pointeur sur la chaîne "Enter X:":
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; X0=pointeur sur la chaîne "Enter X:"
; l'afficher:
        bl      puts
; charger le pointeur sur la chaîne "%d":
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
; trouver de l'espace dans la structure de pile pour la variable "x"
    (X1=FP+28):
```

```
22         add     x1, x29, 28
23  ; X1=adresse de la variable "x"
24  ; passer l'adresse de scanf() et l'appeler:
25         bl      __isoc99_scanf
26  ; charger la valeur 32-bit de la variable dans la partie de pile:
27         ldr     w1, [x29,28]
28  ; W1=x
29  ; charger le pointeur sur la chaîne "You entered %d...\n"
30  ; printf() va prendre la chaîne de texte de X0 et de la variable "x" de X1
      (ou W1)
31         adrp    x0, .LC2
32         add     x0, x0, :lo12:.LC2
33         bl      printf
34  ; retourner 0
35         mov     w0, 0
36  ; restaurer FP et LR, puis ajouter 32 à SP:
37         ldp     x29, x30, [sp], 32
38         ret
```

Il y a 32 octets alloués pour la structure de pile, ce qui est plus que nécessaire. Peut-être dans un soucis d'alignement de mémoire? La partie la plus intéressante est de trouver de l'espace pour la variable x dans la structure de pile (ligne 22). Pourquoi 28? Pour une certaine raison, le compilateur a décidé de stocker cette variable à la fin de la structure de pile locale au lieu du début. L'adresse est passée à `scanf()`, qui stocke l'entrée de l'utilisateur en mémoire à cette adresse. Il s'agit d'une valeur sur 32-bit de type *int*. La valeur est prise à la ligne 27 puis passée à `printf()`.

MIPS

Une place est allouée sur la pile locale pour la variable x, et elle doit être appelée par $\$sp + 24$.

Son adresse est passée à `scanf()`, et la valeur entrée par l'utilisateur est chargée en utilisant l'instruction `LW` («Load Word»), puis passée à `printf()`.

Listing 1.74 : GCC 4.4.5 avec optimisation (résultat en sortie de l'assembleur)

```
$LC0:
        .ascii  "Enter X:\000"
$LC1:
        .ascii  "%d\000"
$LC2:
        .ascii  "You entered %d...\012\000"
main:
; prologue de la fonction:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-40
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,36($sp)
; appel de puts():
        lw      $25,%call16(puts)($28)
        lui     $4,%hi($LC0)
        jalr    $25
```

```
        addiu   $4,$4,%lo($LC0) ; slot de délai de branchement
; appel de scanf():
        lw      $28,16($sp)
        lui     $4,%hi($LC1)
        lw      $25,%call16(__isoc99_scanf)($28)
; définir le 2nd argument de scanf(), $a1=$sp+24:
        addiu   $5,$sp,24
        jalr    $25
        addiu   $4,$4,%lo($LC1) ; slot de délai de branchement

; appel de printf():
        lw      $28,16($sp)
; définir le 2nd argument de printf(),
; charger un mot à l'adresse $sp+24:
        lw      $5,24($sp)
        lw      $25,%call16(printf)($28)
        lui     $4,%hi($LC2)
        jalr    $25
        addiu   $4,$4,%lo($LC2) ; slot de délai de branchement

; épilogue de la fonction:
        lw      $31,36($sp)
; mettre la valeur de retour à 0:
        move    $2,$0
; retourner:
        j       $31
        addiu   $sp,$sp,40      ; slot de délai de branchement
```

IDA affiche la disposition de la pile comme suit:

Listing 1.75 : GCC 4.4.5 avec optimisation (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_18   = -0x18
.text:00000000 var_10   = -0x10
.text:00000000 var_4    = -4
.text:00000000
; prologue de la fonction:
.text:00000000          lui     $gp, (__gnu_local_gp >> 16)
.text:00000004          addiu   $sp, -0x28
.text:00000008          la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C          sw      $ra, 0x28+var_4($sp)
.text:00000010          sw      $gp, 0x28+var_18($sp)
; appel de puts():
.text:00000014          lw      $t9, (puts & 0xFFFF)($gp)
.text:00000018          lui     $a0, ($LC0 >> 16)  # "Enter X:"
.text:0000001C          jalr    $t9
.text:00000020          la      $a0, ($LC0 & 0xFFFF)  # "Enter X:" ; slot de
    délai de branchement
; appel de scanf():
.text:00000024          lw      $gp, 0x28+var_18($sp)
.text:00000028          lui     $a0, ($LC1 >> 16)  # "%d"
.text:0000002C          lw      $t9, (__isoc99_scanf & 0xFFFF)($gp)
```

```
; définir le 2nd argument de scanf(), $a1=$sp+24:
.text:00000030          addiu   $a1, $sp, 0x28+var_10
.text:00000034          jalr    $t9  ; slot de délai de branchement
.text:00000038          la      $a0, ($LC1 & 0xFFFF)  # "%d"
; appel de printf():
.text:0000003C          lw      $gp, 0x28+var_18($sp)
; définir le 2nd argument de printf(),
; charger un mot à l'adresse $sp+24:
.text:00000040          lw      $a1, 0x28+var_10($sp)
.text:00000044          lw      $t9, (printf & 0xFFFF)($gp)
.text:00000048          lui     $a0, ($LC2 >> 16)  # "You entered %d...\n"
.text:0000004C          jalr    $t9
.text:00000050          la      $a0, ($LC2 & 0xFFFF)  # "You entered %d...\n"
    ; slot de délai de branchement
; épilogue de la fonction:
.text:00000054          lw      $ra, 0x28+var_4($sp)
; mettre la valeur de retour à 0:
.text:00000058          move    $v0, $zero
; retourner:
.text:0000005C          jr      $ra
.text:00000060          addiu   $sp, 0x28 ; slot de délai de branchement
```

1.12.2 Erreur courante

C'est une erreur très courante (et/ou une typo) de passer la valeur de *x* au lieu d'un pointeur sur *x* :

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        scanf ("%d", x); // BUG

        printf ("You entered %d...\n", x);

        return 0;
};
```

Donc que se passe-t-il ici? *x* n'est pas initialisée et contient des données aléatoires de la pile locale. Lorsque `scanf()` est appelée, elle prend la chaîne de l'utilisateur, la convertit en nombre et essaye de l'écrire dans *x*, la considérant comme une adresse en mémoire. Mais il s'agit de bruit aléatoire, donc `scanf()` va essayer d'écrire à une adresse aléatoire. Très probablement, le processus va planter.

Assez intéressant, certaines bibliothèques CRT compilées en debug, mettent un signe distinctif lors de l'allocation de la mémoire, comme 0xCCCCCCCC ou 0x0BADF00D etc. Dans ce cas, *x* peut contenir 0xCCCCCCCC, et `scanf()` va essayer d'écrire à l'adresse 0xCCCCCCCC. Et si vous remarquez que quelque chose dans votre processus essaye d'écrire à l'adresse 0xCCCCCCCC, vous saurez qu'une variable non

initialisée (ou un pointeur) a été utilisée sans initialisation préalable. C'est mieux que si la mémoire nouvellement allouée est juste mise à zéro.

1.12.3 Variables globales

Que se passe-t-il si la variable x de l'exemple précédent n'est pas locale mais globale? Alors, elle sera accessible depuis n'importe quel point, plus seulement depuis le corps de la fonction. Les variables globales sont considérées comme un anti-pattern, mais dans un but d'expérience, nous pouvons le faire.

```
#include <stdio.h>

// maintenant x est une variable globale
int x;

int main()
{
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

MSVC: x86

```
_DATA    SEGMENT
COMM    _x:DWORD
$SG2456    DB    'Enter X:', 0aH, 00H
$SG2457    DB    '%d', 00H
$SG2458    DB    'You entered %d...', 0aH, 00H
_DATA    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
    push   ebp
    mov    ebp, esp
    push   OFFSET $SG2456
    call   _printf
    add    esp, 4
    push   OFFSET _x
    push   OFFSET $SG2457
    call   _scanf
    add    esp, 8
    mov    eax, DWORD PTR _x
    push   eax
    push   OFFSET $SG2458
```

```
    call    _printf
    add     esp, 8
    xor     eax, eax
    pop     ebp
    ret     0
_main    ENDP
_TEXT    ENDS
```

Dans ce cas, la variable x est définie dans la section _DATA et il n'y a pas de mémoire allouée sur la pile locale. Elle est accédée directement, pas par la pile. Les variables globales non initialisées ne prennent pas de place dans le fichier exécutable (en effet, pourquoi aurait-on besoin d'allouer de l'espace pour des variables initialement mises à zéro ?), mais lorsque quelqu'un accède à leur adresse, l'OS va y allouer un bloc de zéros[73].

Maintenant, assignons explicitement une valeur à la variable:

```
int x=10; // valeur par défaut
```

Nous obtenons:

```
_DATA   SEGMENT
_x      DD      0aH

...
```

Ici nous voyons une valeur 0xA de type DWORD (DD signifie DWORD = 32 bit) pour cette variable.

Si vous ouvrez le .exe compilé dans IDA, vous pouvez voir la variable *x* placée au début du segment _DATA, et après elle vous pouvez voir la chaîne de texte.

Si vous ouvrez le .exe compilé de l'exemple précédent dans IDA, où la valeur de *x* n'était pas mise, vous verrez quelque chose comme ça:

Listing 1.76 : IDA

```
.data:0040FA80 _x               dd ?     ; DATA XREF: _main+10
.data:0040FA80                           ; _main+22
.data:0040FA84 dword_40FA84     dd ?     ; DATA XREF: _memset+1E
.data:0040FA84                           ; unknown_libname_1+28
.data:0040FA88 dword_40FA88     dd ?     ; DATA XREF: ___sbh_find_block+5
.data:0040FA88                           ; ___sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem            dd ?     ; DATA XREF: ___sbh_find_block+B
.data:0040FA8C                           ; ___sbh_free_block+2CA
.data:0040FA90 dword_40FA90     dd ?     ; DATA XREF: _V6_HeapAlloc+13
.data:0040FA90                           ; __calloc_impl+72
.data:0040FA94 dword_40FA94     dd ?     ; DATA XREF: ___sbh_free_block+2FE
```

_x est marquée avec ? avec le reste des variables qui ne doivent pas être initialisées. Ceci implique qu'après avoir chargé le .exe en mémoire, de l'espace pour toutes

[73] C'est comme ça que se comportent les VM

ces variables doit être alloué et rempli avec des zéros [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.8p10]. Mais dans le fichier .exe, ces variables non initialisées n'occupent rien du tout. C'est pratique pour les gros tableaux, par exemple.

MSVC: x86 + OllyDbg

Les choses sont encore plus simple ici:

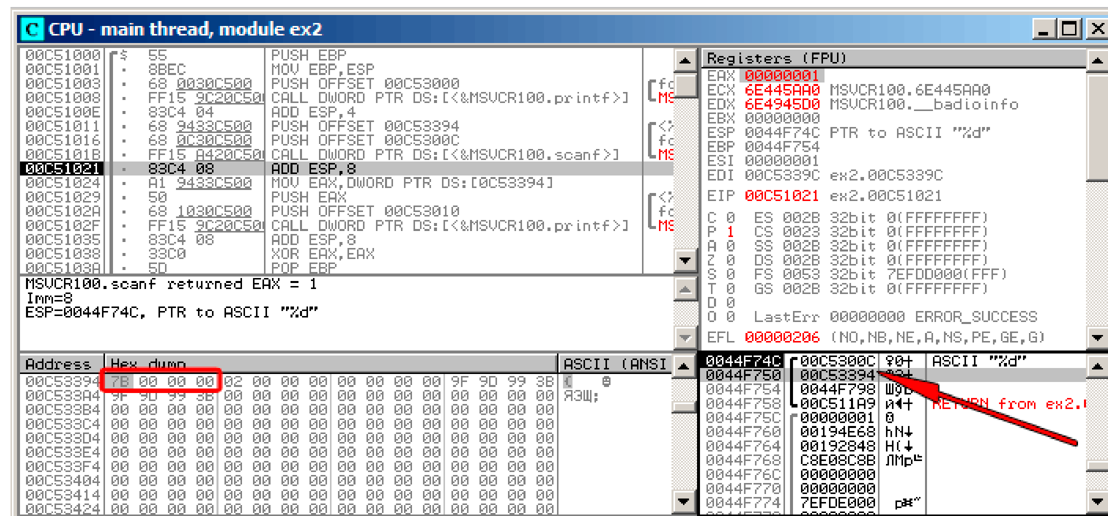


Fig. 1.16: OllyDbg : après l'exécution de `scanf()`

La variable se trouve dans le segment de données. Après que l'instruction PUSH (pousser l'adresse de x) ait été exécutée, l'adresse apparaît dans la fenêtre de la pile. Cliquer droit sur cette ligne et choisir «Follow in dump». La variable va apparaître dans la fenêtre de la mémoire sur la gauche. Après que nous ayons entré 123 dans la console, `0x7B` apparaît dans la fenêtre de la mémoire (voir les régions surlignées dans la copie d'écran).

Mais pourquoi est-ce que le premier octet est `7B`? Logiquement, Il devrait y avoir `00 00 00 7B` ici. La cause de ceci est référé comme endianness, et x86 utilise *little-endian*. Cela implique que l'octet le plus faible poids est écrit en premier, et le plus fort en dernier. Voir à ce propos: 2.2 on page 586. Revenons à l'exemple, la valeur 32-bit est chargée depuis son adresse mémoire dans EAX et passée à `printf()`.

L'adresse mémoire de x est `0x00C53394`.

Dans OllyDbg nous pouvons examiner l'espace mémoire du processus (Alt-M) et nous pouvons voir que cette adresse se trouve dans le segment PE `.data` de notre programme:

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00070000	00067000				Map	R	R	C:\Windows\System32\locale.nl
00190000	00005000			Heap	Priv	RW	RW	
00209000	00007000				Priv	RW Gua	RW Gua	
0044C000	00001000				Priv	RW Gua	RW Gua	
0044D000	00003000			Stack of main thread	Priv	RW	RW	
00590000	00007000				Priv	RW	RW	
00750000	0000C000			Default heap	Priv	RW	RW	
00C50000	00001000	ex2		PE header	Img	R	RWE Cop	
00C51000	00001000	ex2	.text	Code	Img	R E	RWE Cop	
00C52000	00001000	ex2	.rdata	Imports	Img	R	RWE Cop	
00C53000	00001000	ex2	.data	Data	Img	RW	RWE Cop	
00C54000	00001000	ex2	.reloc	Relocations	Img	R	RWE Cop	
6E3E0000	00001000	MSVCR100		PE header	Img	R	RWE Cop	
6E3E1000	000B2000	MSVCR100	.text	Code, imports, exports	Img	R E	RWE Cop	
6E493000	00006000	MSVCR100	.data	Data	Img	RW Cop	RWE Cop	
6E499000	00001000	MSVCR100	.rsrc	Resources	Img	R	RWE Cop	
6E49A000	00005000	MSVCR100	.reloc	Relocations	Img	R	RWE Cop	
755D0000	00001000	Mod_755D		PE header	Img	R	RWE Cop	
755D1000	00003000				Img	R E	RWE Cop	
755D4000	00001000				Img	RW	RWE Cop	
755D5000	00003000				Img	R	RWE Cop	
755E0000	00001000	Mod_755E		PE header	Img	R	RWE Cop	
755E1000	0004D000				Img	R E	RWE Cop	
7562E000	00005000				Img	RW Cop	RWE Cop	
75633000	00009000				Img	R	RWE Cop	
75640000	00001000	Mod_7564		PE header	Img	R	RWE Cop	
75641000	00038000				Img	R E	RWE Cop	
75679000	00002000				Img	RW	RWE Cop	
7567B000	00004000				Img	R	RWE Cop	
76F50000	00010000	kernel32		PE header	Img	R	RWE Cop	
76F60000	000D0000	kernel32	.text	Code, imports, exports	Img	R E	RWE Cop	
77030000	00010000	kernel32	.data	Data	Img	RW Cop	RWE Cop	
77040000	00010000	kernel32	.rsrc	Resources	Img	R	RWE Cop	
77050000	0000B000	kernel32	.reloc	Relocations	Img	R	RWE Cop	
77810000	00001000	KERNELBASE		PE header	Img	R	RWE Cop	
77811000	00040000	KERNELBASE	.text	Code, imports, exports	Img	R E	RWE Cop	
77851000	00002000	KERNELBASE	.data	Data	Img	RW	RWE Cop	
77853000	00001000	KERNELBASE	.rsrc	Resources	Img	R	RWE Cop	
77854000	00003000	KERNELBASE	.reloc	Relocations	Img	R	RWE Cop	
77B20000	00001000	Mod_77B2		PE header	Img	R	RWE Cop	
77B21000	00102000				Img	R E	RWE Cop	
77C23000	0002F000				Img	R	RWE Cop	
77C52000	0000C000				Img	RW Cop	RWE Cop	
77C5E000	0006B000				Img	R	RWE Cop	
77D00000	00001000	ntdll		PE header	Img	R	RWE Cop	
77D10000	000D6000	ntdll	.text	Code, exports	Img	R E	RWE Cop	
77DF0000	00001000	ntdll	RT	Code	Img	R E	RWE Cop	
77E00000	00009000	ntdll	.data	Data	Img	RW Cop	RWE Cop	

Fig. 1.17: OllyDbg : espace mémoire du processus

GCC: x86

Le schéma sous Linux est presque le même, avec la différence que les variables non initialisées se trouvent dans le segment `_bss`. Dans un fichier ELF[74] ce segment possède les attributs suivants:

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

Si toutefois vous initialisez la variable avec une valeur quelconque, e.g. 10, elle sera placée dans le segment `_data`, qui possède les attributs suivants:

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

[74] Executable and Linkable Format: Format de fichier exécutable couramment utilisé sur les systèmes *NIX, Linux inclus

MSVC: x64

Listing 1.77 : MSVC 2012 x64

```
_DATA   SEGMENT
COMM    x:DWORD
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2925 DB      '%d', 00H
$SG2926 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
main    PROC
$LN3:
        sub     rsp, 40

        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, OFFSET FLAT:x
        lea     rcx, OFFSET FLAT:$SG2925 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x
        lea     rcx, OFFSET FLAT:$SG2926 ; 'You entered %d...'
        call    printf

        ; retourner 0
        xor     eax, eax

        add     rsp, 40
        ret     0
main    ENDP
_TEXT   ENDS
```

Le code est presque le même qu'en x86. Notez toutefois que l'adresse de la variable x est passée à `scanf()` en utilisant une instruction LEA, tandis que la valeur de la variable est passée au second `printf()` en utilisant une instruction `MOV`. `DWORD PTR`—fait partie du langage d'assemblage (aucune relation avec le code machine), indique que la taille de la variable est 32-bit et que l'instruction `MOV` doit être encodée en conséquence.

ARM: avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.78 : IDA

```
.text:00000000 ; Segment type: Pure code
.text:00000000         AREA .text, CODE
...
.text:00000000 main
.text:00000000         PUSH    {R4,LR}
.text:00000002         ADR     R0, aEnterX  ; "Enter X:\n"
.text:00000004         BL      __2printf
.text:00000008         LDR     R1, =x
.text:0000000A         ADR     R0, aD       ; "%d"
```

```
.text:0000000C          BL      __0scanf
.text:00000010          LDR     R0, =x
.text:00000012          LDR     R1, [R0]
.text:00000014          ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
.text:00000016          BL      __2printf
.text:0000001A          MOVS    R0, #0
.text:0000001C          POP     {R4,PC}
...
.text:00000020 aEnterX DCB "Enter X:",0xA,0 ; DATA XREF: main+2
.text:0000002A          DCB    0
.text:0000002B          DCB    0
.text:0000002C off_2C   DCD x               ; DATA XREF: main+8
.text:0000002C                              ; main+10
.text:00000030 aD       DCB "%d",0          ; DATA XREF: main+A
.text:00000033          DCB    0
.text:00000034 aYouEnteredD___ DCB "You entered %d...",0xA,0 ; DATA XREF:
    main+14
.text:00000047          DCB 0
.text:00000047 ; .text ends
.text:00000047
...
.data:00000048 ; Segment type: Pure data
.data:00000048          AREA .data, DATA
.data:00000048          ; ORG 0x48
.data:00000048          EXPORT x
.data:00000048 x        DCD 0xA             ; DATA XREF: main+8
.data:00000048                              ; main+10
.data:00000048 ; .data ends
```

Donc, la variable x est maintenant globale, et pour cette raison, elle se trouve dans un autre segment, appelé le segment de données (*.data*). On pourrait demander pour quoi les chaînes de textes sont dans le segment de code (*.text*) et x là. C'est parque c'est une variable et que par définition sa valeur peut changer. En outre, elle peut même changer souvent. Alors que les chaînes de texte ont un type constant, elles ne changent pas, donc elles sont dans le segment *.text*.

Le segment de code peut parfois se trouver dans la ROM[75] d'un circuit (gardez à l'esprit que nous avons maintenant affaire avec de l'électronique embarquée, et que la pénurie de mémoire y est courante), et les variables —en RAM.

Il n'est pas très économique de stocker des constantes en RAM quand vous avez de la ROM.

En outre, les variables en RAM doivent être initialisées, car après le démarrage, la RAM, évidemment, contient des données aléatoires.

En avançant, nous voyons un pointeur sur la variable x (`off_2C`) dans le segment de code, et que toutes les opérations avec cette variable s'effectuent via ce pointeur.

Car la variable x peut se trouver loin de ce morceau de code, donc son adresse doit être sauvée proche du code.

[75] Mémoire morte

L'instruction LDR en mode Thumb ne peut adresser des variables que dans un intervalle de 1020 octets de son emplacement.

et en mode ARM —l'intervalle des variables est de ± 4095 octets.

Et donc l'adresse de la variable x doit se trouver quelque part de très proche, car il n'y a pas de garantie que l'éditeur de liens pourra stocker la variable proche du code, elle peut même se trouver sur un module de mémoire externe.

Encore une chose: si une variable est déclarée comme *const*, le compilateur Keil va l'allouer dans le segment `.constdata`.

Peut-être qu'après, l'éditeur de liens mettra ce segment en ROM aussi, à côté du segment de code.

ARM64

Listing 1.79 : GCC 4.9.1 ARM64 sans optimisation

```
        .comm   x,4,4
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"
f5:
; sauver FP et LR dans la structure de pile locale:
        stp     x29, x30, [sp, -16]!
; définir la pile locale (FP=SP)
        add     x29, sp, 0
; charger le pointeur sur la chaîne "Enter X:":
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        bl      puts
; charger le pointeur sur la chaîne "%d":
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
; générer l'adresse de la variable globale x:
        adrp    x1, x
        add     x1, x1, :lo12:x
        bl      __isoc99_scanf
; générer à nouveau l'adresse de la variable globale x:
        adrp    x0, x
        add     x0, x0, :lo12:x
; charger la valeur de la mémoire à cette adresse:
        ldr     w1, [x0]
; charger le pointeur sur la chaîne "You entered %d...\n":
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        bl      printf
; retourner 0
        mov     w0, 0
; restaurer FP et LR:
```

```
36          ldp     x29, x30, [sp], 16
37          ret
```

Dans ce car la variable x est déclarée comme étant globale et son adresse est calculée en utilisant la paire d'instructions ADRP/ADD (lignes 21 et 25).

MIPS

Variable globale non initialisée

Donc maintenant, la variable x est globale. Compilons en un exécutable plutôt qu'un fichier objet et chargeons-le dans IDA. IDA affiche la variable x dans la section ELF .sbss (vous vous rappelez du «Pointeur Global» ? 1.5.4 on page 34), puisque cette variable n'est pas initialisée au début.

Listing 1.80 : GCC 4.4.5 avec optimisation (IDA)

```
.text:004006C0 main:
.text:004006C0
.text:004006C0 var_10  = -0x10
.text:004006C0 var_4   = -4
.text:004006C0
; prologue de la fonction:
.text:004006C0         lui     $gp, 0x42
.text:004006C4         addiu   $sp, -0x20
.text:004006C8         li      $gp, 0x418940
.text:004006CC         sw      $ra, 0x20+var_4($sp)
.text:004006D0         sw      $gp, 0x20+var_10($sp)
; appel de puts():
.text:004006D4         la      $t9, puts
.text:004006D8         lui     $a0, 0x40
.text:004006DC         jalr    $t9 ; puts
.text:004006E0         la      $a0, aEnterX     # "Enter X:" ; slot de délai de branchement
; appel de scanf():
.text:004006E4         lw      $gp, 0x20+var_10($sp)
.text:004006E8         lui     $a0, 0x40
.text:004006EC         la      $t9, __isoc99_scanf
; préparer l'adresse de x:
.text:004006F0         la      $a1, x
.text:004006F4         jalr    $t9 ; __isoc99_scanf
.text:004006F8         la      $a0, aD          # "%d" ; slot de délai de branchement
; appel de printf():
.text:004006FC         lw      $gp, 0x20+var_10($sp)
.text:00400700         lui     $a0, 0x40
; prendre l'adresse de x:
.text:00400704         la      $v0, x
.text:00400708         la      $t9, printf
; charger la valeur de la variable "x" et la passer à printf() dans $a1:
.text:0040070C         lw      $a1, (x - 0x41099C)($v0)
.text:00400710         jalr    $t9 ; printf
```

```
.text:00400714         la      $a0, aYouEnteredD___  # "You entered %d...\n"
    ; slot de délai de branchement
; épilogue de la fonction:
.text:00400718         lw      $ra, 0x20+var_4($sp)
.text:0040071C         move    $v0, $zero
.text:00400720         jr      $ra
.text:00400724         addiu   $sp, 0x20  ; slot de délai de branchement

...

.sbss:0041099C  # Type de segment: Non initialisé
.sbss:0041099C         .sbss
.sbss:0041099C         .globl x
.sbss:0041099C x:      .space 4
.sbss:0041099C
```

IDA réduit le volume des informations, donc nous allons générer un listing avec objdump et le commenter:

Listing 1.81 : GCC 4.4.5 avec optimisation (objdump)

```
004006c0 <main>:
; prologue de la fonction:
  4006c0:  3c1c0042  lui     gp,0x42
  4006c4:  27bdffe0  addiu   sp,sp,-32
  4006c8:  279c8940  addiu   gp,gp,-30400
  4006cc:  afbf001c  sw      ra,28(sp)
  4006d0:  afbc0010  sw      gp,16(sp)
; appel de puts():
  4006d4:  8f998034  lw      t9,-32716(gp)
  4006d8:  3c040040  lui     a0,0x40
  4006dc:  0320f809  jalr    t9
  4006e0:  248408f0  addiu   a0,a0,2288 ; slot de délai de branchement
; appel de scanf():
  4006e4:  8fbc0010  lw      gp,16(sp)
  4006e8:  3c040040  lui     a0,0x40
  4006ec:  8f998038  lw      t9,-32712(gp)
; préparer l'adresse de x:
  4006f0:  8f858044  lw      a1,-32700(gp)
  4006f4:  0320f809  jalr    t9
  4006f8:  248408fc  addiu   a0,a0,2300 ; slot de délai de branchement
; appel de printf():
  4006fc:  8fbc0010  lw      gp,16(sp)
  400700:  3c040040  lui     a0,0x40
; prendre l'adresse de x:
  400704:  8f828044  lw      v0,-32700(gp)
  400708:  8f99803c  lw      t9,-32708(gp)
; charger la valeur de la variable "x" et la passer à printf() dans $a1:
  40070c:  8c450000  lw      a1,0(v0)
  400710:  0320f809  jalr    t9
  400714:  24840900  addiu   a0,a0,2304 ; slot de délai de branchement
; épilogue de la fonction:
  400718:  8fbf001c  lw      ra,28(sp)
  40071c:  00001021  move    v0,zero
```

```
34   400720:  03e00008  jr      ra
35   400724:  27bd0020  addiu   sp,sp,32   ; slot de délai de branchement
36 ; groupe de NOPs servant à aligner la prochaine fonction sur un bloc de
     16-octet:
37   400728:  00200825  move    at,at
38   40072c:  00200825  move    at,at
```

Nous voyons maintenant que l'adresse de la variable x est lue depuis un buffer de 64KiB en utilisant GP et en lui ajoutant un offset négatif (ligne 18). Plus que ça, les adresses des trois fonctions externes qui sont utilisées dans notre exemple (`puts()`, `scanf()`, `printf()`), sont aussi lues depuis le buffer de données globale en utilisant GP (lignes 9, 16 et 26). GP pointe sur le milieu du buffer, et de tels offsets suggèrent que les adresses des trois fonctions, et aussi l'adresse de la variable x, sont toutes stockées quelque part au début du buffer. Cela fait du sens, car notre exemple est minuscule.

Une autre chose qui mérite d'être mentionnée est que la fonction se termine avec deux NOPs (`MOVE $AT,$AT` — une instruction sans effet), afin d'aligner le début de la fonction suivante sur un bloc de 16-octet.

Variable globale initialisée

Modifions notre exemple en affectant une valeur par défaut à la variable x :

```
int x=10; // valeur par défaut
```

Maintenant IDA montre que la variable x se trouve dans la section .data:

Listing 1.82 : GCC 4.4.5 avec optimisation (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_10 = -0x10
.text:004006A0 var_8  = -8
.text:004006A0 var_4  = -4
.text:004006A0
.text:004006A0        lui     $gp, 0x42
.text:004006A4        addiu   $sp, -0x20
.text:004006A8        li      $gp, 0x418930
.text:004006AC        sw      $ra, 0x20+var_4($sp)
.text:004006B0        sw      $s0, 0x20+var_8($sp)
.text:004006B4        sw      $gp, 0x20+var_10($sp)
.text:004006B8        la      $t9, puts
.text:004006BC        lui     $a0, 0x40
.text:004006C0        jalr    $t9 ; puts
.text:004006C4        la      $a0, aEnterX     # "Enter X:"
.text:004006C8        lw      $gp, 0x20+var_10($sp)
; préparer la partie haute de l'adresse de x:
.text:004006CC        lui     $s0, 0x41
.text:004006D0        la      $t9, __isoc99_scanf
.text:004006D4        lui     $a0, 0x40
; et ajouter la partie basse de l'adresse de x:
```

```
.text:004006D8        addiu   $a1, $s0, (x - 0x410000)
; maintenant l'adresse de x est dans $a1.
.text:004006DC        jalr    $t9 ; __isoc99_scanf
.text:004006E0        la      $a0, aD          # "%d"
.text:004006E4        lw      $gp, 0x20+var_10($sp)
; prendre un mot dans la mémoire:
.text:004006E8        lw      $a1, x
; la valeur de x est maintenant dans $a1.
.text:004006EC        la      $t9, printf
.text:004006F0        lui     $a0, 0x40
.text:004006F4        jalr    $t9 ; printf
.text:004006F8        la      $a0, aYouEnteredD___  # "You entered %d...\n"
.text:004006FC        lw      $ra, 0x20+var_4($sp)
.text:00400700        move    $v0, $zero
.text:00400704        lw      $s0, 0x20+var_8($sp)
.text:00400708        jr      $ra
.text:0040070C        addiu   $sp, 0x20

...

.data:00410920        .globl x
.data:00410920 x:     .word 0xA
```

Pourquoi pas .sdata? Peut-être que cela dépend d'une option de GCC?

Néanmoins, maintenant x est dans .data, qui une zone mémoire générale, et nous pouvons regarder comment y travailler avec des variables.

L'adresse de la variable doit être formée en utilisant une paire d'instructions.

Dans notre cas, ce sont LUI («Load Upper Immediate») et ADDIU («Add Immediate Unsigned Word»).

Voici le listing d'objdump pour y regarder de plus près:

Listing 1.83 : GCC 4.4.5 avec optimisation (objdump)

```
004006a0 <main>:
  4006a0:  3c1c0042  lui     gp,0x42
  4006a4:  27bdffe0  addiu   sp,sp,-32
  4006a8:  279c8930  addiu   gp,gp,-30416
  4006ac:  afbf001c  sw      ra,28(sp)
  4006b0:  afb00018  sw      s0,24(sp)
  4006b4:  afbc0010  sw      gp,16(sp)
  4006b8:  8f998034  lw      t9,-32716(gp)
  4006bc:  3c040040  lui     a0,0x40
  4006c0:  0320f809  jalr    t9
  4006c4:  248408d0  addiu   a0,a0,2256
  4006c8:  8fbc0010  lw      gp,16(sp)
; préparer la partie haute de l'adresse de x:
  4006cc:  3c100041  lui     s0,0x41
  4006d0:  8f998038  lw      t9,-32712(gp)
  4006d4:  3c040040  lui     a0,0x40
; ajouter la partie basse de l'adresse de x:
  4006d8:  26050920  addiu   a1,s0,2336
```

```
; maintenant l'adresse de x est dans $a1.
  4006dc:  0320f809  jalr    t9
  4006e0:  248408dc  addiu   a0,a0,2268
  4006e4:  8fbc0010  lw      gp,16(sp)
; la partie haute de l'adresse de x est toujours dans $s0.
; lui ajouter la partie basse et charger un mot de la mémoire:
  4006e8:  8e050920  lw      a1,2336(s0)
; la valeur de x est maintenant dans $a1.
  4006ec:  8f99803c  lw      t9,-32708(gp)
  4006f0:  3c040040  lui     a0,0x40
  4006f4:  0320f809  jalr    t9
  4006f8:  248408e0  addiu   a0,a0,2272
  4006fc:  8fbf001c  lw      ra,28(sp)
  400700:  00001021  move    v0,zero
  400704:  8fb00018  lw      s0,24(sp)
  400708:  03e00008  jr      ra
  40070c:  27bd0020  addiu   sp,sp,32
```

Nous voyons que l'adresse est formée en utilisant LUI et ADDIU, mais la partie haute de l'adresse est toujours dans le registre $S0, et il est possible d'encoder l'offset en une instruction LW («Load Word »), donc une seule instruction LW est suffisante pour charger une valeur de la variable et la passer à printf().

Les registres contenant des données temporaires sont préfixés avec T-, mais ici nous en voyons aussi qui sont préfixés par S-, leur contenu doit être doit être sauvegardé quelque part avant de les utiliser dans d'autres fonctions.

C'est pourquoi la valeur de $S0 a été mise à l'adresse 0x4006cc et utilisée de nouveau à l'adresse 0x4006e8, après l'appel de scanf(). La fonction scanf() ne change pas cette valeur.

1.12.4 scanf()

Comme il a déjà été écrit, il est plutôt dépassé d'utiliser scanf() aujourd'hui. Mais si nous devons, il faut vérifier si scanf() se termine correctement sans erreur.

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        if (scanf ("%d", &x)==1)
                printf ("You entered %d...\n", x);
        else
                printf ("What you entered? Huh?\n");

        return 0;
};
```

Par norme, la fonction `scanf()`[76] renvoie le nombre de champs qui ont été lus avec succès.

Dans notre cas, si tout se passe bien et que l'utilisateur entre un nombre `scanf()` renvoie 1, ou en cas d'erreur (ou EOF[77]) — 0.

Ajoutons un peu de code C pour vérifier la valeur de retour de `scanf()` et afficher un message d'erreur en cas d'erreur.

Cela fonctionne comme attendu:

```
C:\...>ex3.exe
Enter X:
123
You entered 123...

C:\...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

MSVC: x86

Voici ce que nous obtenons dans la sortie assembleur (MSVC 2010) :

```
        lea     eax, DWORD PTR _x$[ebp]
        push    eax
        push    OFFSET $SG3833 ; '%d', 00H
        call    _scanf
        add     esp, 8
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        push    OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
        call    _printf
        add     esp, 8
        jmp     SHORT $LN1@main
$LN2@main:
        push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
        call    _printf
        add     esp, 4
$LN1@main:
        xor     eax, eax
```

La fonction appelante (`main()`) a besoin du résultat de la fonction appelée, donc la fonction appelée le renvoie dans la registre EAX.

Nous le vérifions avec l'aide de l'instruction `CMP EAX, 1` (*CoMPare*). En d'autres mots, nous comparons la valeur dans le registre EAX avec 1.

[76] scanf, wscanf: MSDN
[77] End of File (fin de fichier)

Une instruction de saut conditionnelle JNE suit l'instruction CMP. JNE signifie *Jump if Not Equal* (saut si non égal).

Donc, si la valeur dans le registre EAX n'est pas égale à 1, le CPU va poursuivre l'exécution à l'adresse mentionnée dans l'opérande JNE, dans notre cas $LN2@main. Passer le contrôle à cette adresse résulte en l'exécution par le CPU de printf() avec l'argument What you entered? Huh?. Mais si tout est bon, le saut conditionnel n'est pas pris, et un autre appel à printf() est exécuté, avec deux arguments:
'You entered %d...' et la valeur de x.

Puisque dans ce cas le second printf() n'a pas été exécuté, il y a un JMP qui le précède (saut inconditionnel). Il passe le contrôle au point après le second printf() et juste avant l'instruction XOR EAX, EAX, qui implémente return 0.

Donc, on peut dire que comparer une valeur avec une autre est *usuellement* implémenté par la paire d'instructions CMP/Jcc, où *cc* est un *code de condition*. CMP compare deux valeurs et met les flags[78] du processeur. Jcc vérifie ces flags et décide de passer LE Contrôle à l'adresse spécifiée ou non.

Cela peut sembler paradoxal, mais l'instruction CMP est en fait un SUB (soustraction). Toutes les instructions arithmétiques mettent les flags du processeur, pas seulement CMP. Si nous comparons 1 et 1, $1 - 1$ donne 0 donc le flag ZF va être mis (signifiant que le dernier résultat est 0). Dans aucune autre circonstance ZF ne sera mis, sauf si les opérandes sont égaux. JNE vérifie seulement le flag ZF et saute seulement si il n'est pas mis. JNE est un synonyme pour JNZ (*Jump if Not Zero* (saut si non zéro)). L'assembleur génère le même opcode pour les instructions JNE et JNZ. Donc, l'instruction CMP peut être remplacée par une instruction SUB et presque tout ira bien, à la différence que SUB altère la valeur du premier opérande. CMP est un *SUB sans sauver le résultat, mais modifiant les flags*.

MSVC: x86: IDA

C'est le moment de lancer IDA et d'essayer de faire quelque chose avec. À propos, pour les débutants, c'est une bonne idée d'utiliser l'option /MD de MSVC, qui signifie que toutes les fonctions standards ne vont pas être liées avec le fichier exécutable, mais vont à la place être importées depuis le fichier MSVCR*.DLL. Ainsi il est plus facile de voir quelles fonctions standards sont utilisées et où.

En analysant du code dans IDA, il est très utile de laisser des notes pour soi-même (et les autres). En la circonstance, analysons cet exemple, nous voyons que JNZ sera déclenché en cas d'erreur. Donc il est possible de déplacer le curseur sur le label, de presser «n» et de lui donner le nom «error». Créons un autre label—dans «exit». Voici mon résultat:

```
.text:00401000 _main proc near
.text:00401000
.text:00401000 var_4 = dword ptr -4
.text:00401000 argc  = dword ptr  8
.text:00401000 argv  = dword ptr  0Ch
.text:00401000 envp  = dword ptr  10h
.text:00401000
```

[78] flags x86, voir aussi: Wikipédia.

```
.text:00401000        push    ebp
.text:00401001        mov     ebp, esp
.text:00401003        push    ecx
.text:00401004        push    offset Format ; "Enter X:\n"
.text:00401009        call    ds:printf
.text:0040100F        add     esp, 4
.text:00401012        lea     eax, [ebp+var_4]
.text:00401015        push    eax
.text:00401016        push    offset aD ; "%d"
.text:0040101B        call    ds:scanf
.text:00401021        add     esp, 8
.text:00401024        cmp     eax, 1
.text:00401027        jnz     short error
.text:00401029        mov     ecx, [ebp+var_4]
.text:0040102C        push    ecx
.text:0040102D        push    offset aYou ; "You entered %d...\n"
.text:00401032        call    ds:printf
.text:00401038        add     esp, 8
.text:0040103B        jmp     short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D        push    offset aWhat ; "What you entered? Huh?\n"
.text:00401042        call    ds:printf
.text:00401048        add     esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B        xor     eax, eax
.text:0040104D        mov     esp, ebp
.text:0040104F        pop     ebp
.text:00401050        retn
.text:00401050 _main endp
```

Maintenant, il est légèrement plus facile de comprendre le code. Toutefois, ce n'est pas une bonne idée de commenter chaque instruction.

Vous pouvez aussi cacher (replier) des parties d'une fonction dans IDA. Pour faire cela, marquez le bloc, puis appuyez sur Ctrl-«–» sur le pavé numérique et entrez le texte qui doit être affiché à la place.

Cachons deux blocs et donnons leurs un nom:

```
.text:00401000 _text segment para public 'CODE' use32
.text:00401000        assume cs:_text
.text:00401000        ;org 401000h
.text:00401000 ; ask for X
.text:00401012 ; get X
.text:00401024        cmp  eax, 1
.text:00401027        jnz  short error
.text:00401029 ; print result
.text:0040103B        jmp  short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D        push offset aWhat ; "What you entered? Huh?\n"
.text:00401042        call ds:printf
```

```
.text:00401048       add  esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B       xor  eax, eax
.text:0040104D       mov  esp, ebp
.text:0040104F       pop  ebp
.text:00401050       retn
.text:00401050 _main endp
```

Pour étendre les parties de code précédemment cachées. utilisez Ctrl-«+» sur le pavé numérique.

En appuyant sur «space», nous voyons comment IDA représente une fonction sous forme de graphe:

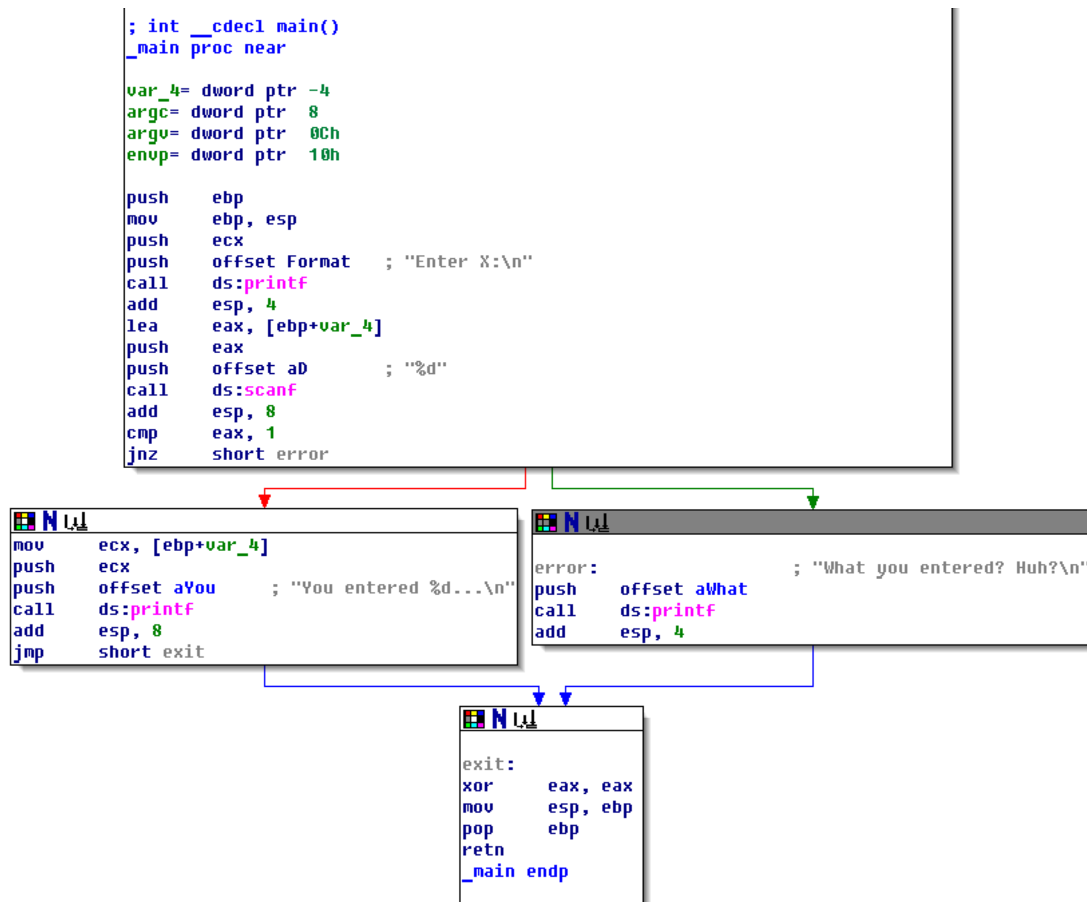


Fig. 1.18: IDA en mode graphe

Il y a deux flèches après chaque saut conditionnel: une verte et une rouge. La flèche verte pointe vers le bloc qui sera exécuté si le saut est déclenché, et la rouge sinon.

Il est possible de replier des nœuds dans ce mode et de leurs donner aussi un nom («group nodes »). Essayons avec 3 blocs:

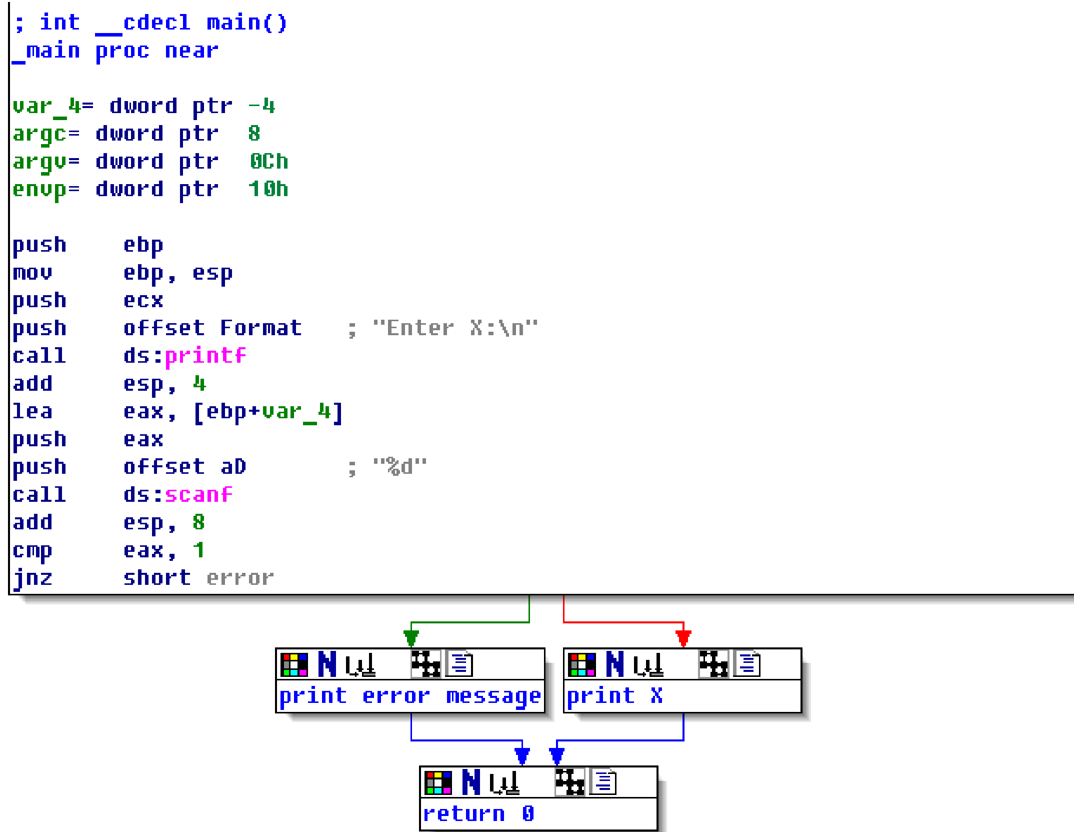


Fig. 1.19: IDA en mode graphe avec 3 nœuds repliés

C'est très pratique. On peut dire qu'une part importante du travail des rétro-ingénieurs (et de tout autre chercheur également) est de réduire la quantité d'information avec laquelle travailler.

MSVC: x86 + OllyDbg

Essayons de hacker notre programme dans OllyDbg, pour le forcer à penser que `scanf()` fonctionne toujours sans erreur. Lorsque l'adresse d'une variable locale est passée à `scanf()`, la variable contient initiallement toujours des restes de données aléatoires, dans ce cas `0x6E494714` :

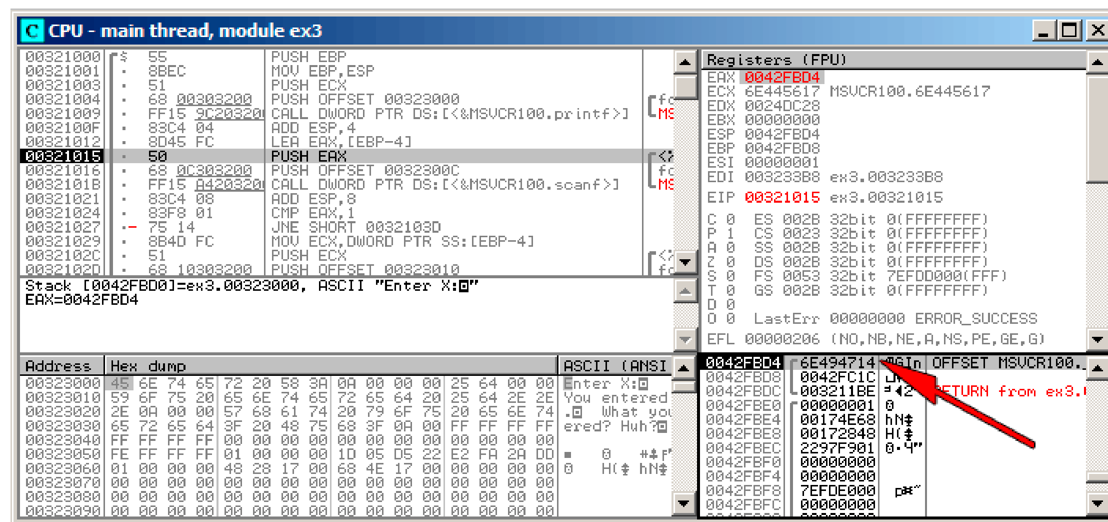


Fig. 1.20: OllyDbg : passer l'adresse de la variable à `scanf()`

Lorsque `scanf()` s'exécute dans la console, entrons quelque chose qui n'est pas du tout un nombre, comme «asdasd». `scanf()` termine avec 0 dans EAX, ce qui indique qu'une erreur s'est produite.

Nous pouvons vérifier la variable locale dans le pile et noter qu'elle n'a pas changé. En effet, qu'aurait écrit `scanf()` ici? Elle n'a simplement rien fait à part renvoyer zéro.

Essayons de «hacker» notre programme. Clique-droit sur EAX, parmi les options il y a «Set to 1» (mettre à 1). C'est ce dont nous avons besoin.

Nous avons maintenant 1 dans EAX, donc la vérification suivante va s'exécuter comme souhaiter et `printf()` va afficher la valeur de la variable dans la pile.

Lorsque nous lançons le programme (F9) nous pouvons voir ceci dans la fenêtre de la console:

Listing 1.84 : fenêtre console

```
Enter X:
asdasd
You entered 1850296084...
```

En effet, 1850296084 est la représentation en décimal du nombre dans la pile (`0x6E494714`)!

MSVC: x86 + Hiew

Cela peut également être utilisé comme un exemple simple de modification de fichier exécutable. Nous pouvons essayer de modifier l'exécutable de telle sorte que le programme va toujours afficher notre entrée, quelle qui'elle soit.

En supposant que l'exécutable est compilé avec la bibliothèque externe `MSVCR*.DLL` (i.e., avec l'option `/MD`) [79], nous voyons la fonction `main()` au début de la section `.text`. Ouvrons l'exécutable dans Hiew et cherchons le début de la section `.text` (Enter, F8, F6, Enter, Enter).

Nous pouvons voir cela:

```
Hiew: ex3.exe
   C:\Polygon\ollydbg\ex3.exe              FRO --------          a32 PE .00401000 Hie
.00401000: 55                           push        ebp
.00401001: 8BEC                         mov         ebp,esp
.00401003: 51                           push        ecx
.00401004: 6800304000                   push        000403000 ;'Enter X:' --1
.00401009: FF1594204000                 call        printf
.0040100F: 83C404                       add         esp,4
.00401012: 8D45FC                       lea         eax,[ebp][-4]
.00401015: 50                           push        eax
.00401016: 680C304000                   push        00040300C --2
.0040101B: FF158C204000                 call        scanf
.00401021: 83C408                       add         esp,8
.00401024: 83F801                       cmp         eax,1
.00401027: 7514                         jnz        .00040103D --3
.00401029: 8B4DFC                       mov         ecx,[ebp][-4]
.0040102C: 51                           push        ecx
.0040102D: 6810304000                   push        000403010 ;'You entered %d...'
.00401032: FF1594204000                 call        printf
.00401038: 83C408                       add         esp,8
.0040103B: EB0E                         jmps       .00040104B --5
.0040103D: 6824304000                  3push        000403024 ;'What you entered?
.00401042: FF1594204000                 call        printf
.00401048: 83C404                       add         esp,4
.0040104B: 33C0                        5xor         eax,eax
.0040104D: 8BE5                         mov         esp,ebp
.0040104F: 5D                           pop         ebp
.00401050: C3                           retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.00401051: B84D5A0000                   mov         eax,000005A4D ;'  ZM'
1Global 2FilBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 9 1byte 10Leave 11Nak
```

Fig. 1.21: Hiew: fonction `main()`

Hiew trouve les chaîne ASCIIZ[80] et les affiche, comme il le fait avec le nom des fonctions importées.

[79] c'est aussi appelé «dynamic linking»

[80] ASCII Zero (chaîne ASCII terminée par un octet nul (à zéro))

Déplacez le curseur à l'adresse .00401027 (où se trouve l'instruction JNZ, que l'on doit sauter), appuyez sur F3, et ensuite tapez «9090» (qui signifie deux NOPs) :

```
Hiew: ex3.exe
   C:\Polygon\ollydbg\ex3.exe                    FWO EDITMODE        a32 PE  0000
00000400: 55                          push        ebp
00000401: 8BEC                        mov         ebp,esp
00000403: 51                          push        ecx
00000404: 6800304000                  push        000403000 ;' @0 '
00000409: FF1594204000                call        d,[000402094]
0000040F: 83C404                      add         esp,4
00000412: 8D45FC                      lea         eax,[ebp][-4]
00000415: 50                          push        eax
00000416: 680C304000                  push        00040300C ;' @0 '
0000041B: FF158C204000                call        d,[00040208C]
00000421: 83C408                      add         esp,8
00000424: 83F801                      cmp         eax,1
00000427: 90                          nop
00000428: 90                          nop
00000429: 8B4DFC                      mov         ecx,[ebp][-4]
0000042C: 51                          push        ecx
0000042D: 6810304000                  push        000403010 ;' @0 '
00000432: FF1594204000                call        d,[000402094]
00000438: 83C408                      add         esp,8
0000043B: EB0E                        jmps        00000044B
0000043D: 6824304000                  push        000403024 ;' @0$'
00000442: FF1594204000                call        d,[000402094]
00000448: 83C404                      add         esp,4
0000044B: 33C0                        xor         eax,eax
0000044D: 8BE5                        mov         esp,ebp
0000044F: 5D                          pop         ebp
00000450: C3                          retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-
1      2NOPs   3      4      5      6      7      8Table  9      10
```

Fig. 1.22: Hiew: remplacement de JNZ par deux NOPs

Appuyez sur F9 (update). Maintenant, l'exécutable est sauvé sur le disque. Il va se comporter comme nous le voulions.

Deux NOPs ne constitue probablement pas l'approche la plus esthétique. Une autre façon de modifier cette instruction est d'écrire simplement 0 dans le second octet de l'opcode ((jump offset), donc ce JNZ va toujours sauter à l'instruction suivante.

Nous pouvons également faire le contraire: remplacer le premier octet avec EB sans modifier le second octet (jump offset). Nous obtiendrions un saut inconditionnel qui est toujours déclenché. Dans ce cas le message d'erreur sera affiché à chaque fois, peu importe l'entrée.

MSVC: x64

Puisque nous travaillons ici avec des variables typées *int*, qui sont toujours 32-bit en x86-64, nous voyons comment la partie 32-bit des registres (préfixés avec E-) est également utilisée ici. Lorsque l'on travaille avec des ponteurs, toutefois, les parties 64-bit des registres sont utilisées, préfixés avec R-.

Listing 1.85 : MSVC 2012 x64

```
_DATA   SEGMENT
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2926 DB      '%d', 00H
$SG2927 DB      'You entered %d...', 0aH, 00H
$SG2929 DB      'What you entered? Huh?', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN5:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2926 ; '%d'
        call    scanf
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2927 ; 'You entered %d...'
        call    printf
        jmp     SHORT $LN1@main
$LN2@main:
        lea     rcx, OFFSET FLAT:$SG2929 ; 'What you entered? Huh?'
        call    printf
$LN1@main:
        ; retourner 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
_TEXT   ENDS
END
```

ARM

ARM: avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.86 : avec optimisation Keil 6/2013 (Mode Thumb)

```
var_8    = -8

         PUSH    {R3,LR}
```

```
        ADR     R0, aEnterX     ; "Enter X:\n"
        BL      __2printf
        MOV     R1, SP
        ADR     R0, aD          ; "%d"
        BL      __0scanf
        CMP     R0, #1
        BEQ     loc_1E
        ADR     R0, aWhatYouEntered ; "What you entered? Huh?\n"
        BL      __2printf

loc_1A                          ; CODE XREF: main+26
        MOVS    R0, #0
        POP     {R3,PC}

loc_1E                          ; CODE XREF: main+12
        LDR     R1, [SP,#8+var_8]
        ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
        BL      __2printf
        B       loc_1A
```

Les nouvelles instructions sont CMP et BEQ[81].

CMP est similaire à l'instruction x86 du même nom, elle soustrait l'un des arguments à l'autre et met à jour les flags si nécessaire.

BEQ saute à une autre adresse si les opérandes étaient égaux l'un à l'autre, ou, si le résultat du dernier calcul était 0, ou si le flag Z est à 1. Elle se comporte comme JZ en x86.

Tout le reste est simple: le flux d'exécution se sépare en deux branches, puis les branches convergent vers le point où 0 est écrit dans le registre R0 comme valeur de retour de la fonction, et cette dernière se termine.

ARM64

Listing 1.87 : GCC 4.9.1 ARM64 sans optimisation

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"
.LC3:
        .string "What you entered? Huh?"
f6:
; sauver FP et LR dans la structure de pile locale:
        stp     x29, x30, [sp, -32]!
; définir la pile locale (FP=SP)
        add     x29, sp, 0
; charger le pointeur sur la chaîne "Enter X:":
```

[81](PowerPC, ARM) Branch if Equal

```
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        bl      puts
; charger le pointeur sur la chaîne "%d":
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
; calculer l'adresse de la variable x dans la pile locale
        add     x1, x29, 28
        bl      __isoc99_scanf
; scanf() renvoie son résultat dans W0.
; le vérifier:
        cmp     w0, 1
; BNE est Branch if Not Equal (branchement si non égal)
; donc if W0<>1, un saut en L2 sera effectué
        bne     .L2
; à ce point W0=1, signifie pas d'erreur
; charger la valeur de x depuis la pile locale
        ldr     w1, [x29,28]
; charger le pointeur sur la chaîne "You entered %d...\n":
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        bl      printf
; sauter le code, qui affiche la chaîne "What you entered? Huh?":
        b       .L3
.L2:
; charger le pointeur sur la chaîne "What you entered? Huh?":
        adrp    x0, .LC3
        add     x0, x0, :lo12:.LC3
        bl      puts
.L3:
; retourner 0
        mov     w0, 0
; restaurer FP et LR:
        ldp     x29, x30, [sp], 32
        ret
```

Dans ce cas, le flux de code se sépare avec l'utilisation de la paire d'instructions CMP/BNE (Branch if Not Equal) (branchement si non égal).

MIPS

Listing 1.88 : avec optimisation GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_18    = -0x18
.text:004006A0 var_10    = -0x10
.text:004006A0 var_4     = -4
.text:004006A0
.text:004006A0           lui     $gp, 0x42
.text:004006A4           addiu   $sp, -0x28
.text:004006A8           li      $gp, 0x418960
.text:004006AC           sw      $ra, 0x28+var_4($sp)
```

```
.text:004006B0          sw      $gp, 0x28+var_18($sp)
.text:004006B4          la      $t9, puts
.text:004006B8          lui     $a0, 0x40
.text:004006BC          jalr    $t9 ; puts
.text:004006C0          la      $a0, aEnterX     # "Enter X:"
.text:004006C4          lw      $gp, 0x28+var_18($sp)
.text:004006C8          lui     $a0, 0x40
.text:004006CC          la      $t9, __isoc99_scanf
.text:004006D0          la      $a0, aD          # "%d"
.text:004006D4          jalr    $t9 ; __isoc99_scanf
.text:004006D8          addiu   $a1, $sp, 0x28+var_10  # branch delay slot
.text:004006DC          li      $v1, 1
.text:004006E0          lw      $gp, 0x28+var_18($sp)
.text:004006E4          beq     $v0, $v1, loc_40070C
.text:004006E8          or      $at, $zero       # branch delay slot, NOP
.text:004006EC          la      $t9, puts
.text:004006F0          lui     $a0, 0x40
.text:004006F4          jalr    $t9 ; puts
.text:004006F8          la      $a0, aWhatYouEntered  # "What you entered?
    Huh?"
.text:004006FC          lw      $ra, 0x28+var_4($sp)
.text:00400700          move    $v0, $zero
.text:00400704          jr      $ra
.text:00400708          addiu   $sp, 0x28

.text:0040070C loc_40070C:
.text:0040070C          la      $t9, printf
.text:00400710          lw      $a1, 0x28+var_10($sp)
.text:00400714          lui     $a0, 0x40
.text:00400718          jalr    $t9 ; printf
.text:0040071C          la      $a0, aYouEnteredD___  # "You entered
    %d...\n"
.text:00400720          lw      $ra, 0x28+var_4($sp)
.text:00400724          move    $v0, $zero
.text:00400728          jr      $ra
.text:0040072C          addiu   $sp, 0x28
```

scanf() renvoie le résultat de son traitement dans le registre $V0. Il est testé à l'adresse 0x004006E4 en comparant la valeur dans $V0 avec celle dans $V1 (1 a été stocké dans $V1 plus tôt, en 0x004006DC). BEQ signifie «Branch Equal» (branchement si égal). Si les deux valeurs sont égales (i.e., succès), l'exécution saute à l'adresse 0x0040070C.

Exercice

Comme nous pouvons voir, les instructions JNE/JNZ peuvent facilement être remplacées par JE/JZ et vice-versa (ou BNE par BEQ et vice-versa). Mais les blocs de base doivent aussi être échangés. Essayez de faire cela pour quelques exemples.

1.12.5 Exercice

- http://challenges.re/53

1.13 Intéressant à noter: variables globales vs. locales

Maintenant vous savez que les variables globales sont remplies avec des zéros par l'OS au début (1.12.3 on page 106, [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.8p10]), mais que les variables locales ne le sont pas.

Parfois, vous avez une variable globale que vous avez oublié d'initialiser et votre programme fonctionne grâce au fait qu'elle est à zéro au début. Puis, vous éditez votre programme et déplacez la variable globale dans une fonction pour la rendre locale. Elle ne sera plus initialisée à zéro et ceci peut résulter en de méchants bogues.

1.14 Accéder aux arguments passés

Maintenant nous savons que la fonction appelante passe les arguments à la fonction appelée par la pile. Mais comment est-ce que la fonction appelée y accède?

Listing 1.89 : exemple simple

```
#include <stdio.h>

int f (int a, int b, int c)
{
        return a*b+c;
};

int main()
{
        printf ("%d\n", f(1, 2, 3));
        return 0;
};
```

1.14.1 x86

MSVC

Voici ce que l'on obtient après compilation (MSVC 2010 Express) :

Listing 1.90 : MSVC 2010 Express

```
_TEXT   SEGMENT
_a$ = 8         ; taille = 4
_b$ = 12        ; taille = 4
_c$ = 16        ; taille = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
```

```
        ret     0
_f      ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        push    3 ; 3ème argument
        push    2 ; 2ème argument
        push    1 ; 1er argument
        call    _f
        add     esp, 12
        push    eax
        push    OFFSET $SG2463 ; '%d', 0aH, 00H
        call    _printf
        add     esp, 8
        ; retourner 0
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Ce que l'on voit, c'est que la fonction `main()` pousse 3 nombres sur la pile et appelle `f(int,int,int)`.

L'accès aux arguments à l'intérieur de `f()` est organisé à l'aide de macros comme: `_a$ = 8`, de la même façon que pour les variables locales, mais avec des offsets positifs (accédés avec *plus*). Donc, nous accédons à la partie *hors* de la structure locale de pile en ajoutant la macro `_a$` à la valeur du registre EBP.

Ensuite, la valeur de a est stockée dans EAX. Après l'exécution de l'instruction IMUL, la valeur de EAX est le produit de la valeur de EAX et du contenu de `_b`.

Après cela, ADD ajoute la valeur dans `_c` à EAX.

La valeur dans EAX n'a pas besoin d'être déplacée/copiée : elle est déjà là où elle doit être. Lors du retour dans la fonction appelante, elle prend la valeur dans EAX et l'utilise comme argument pour `printf()`.

MSVC + OllyDbg

Illustrons ceci dans OllyDbg. Lorsque nous traçons jusqu'à la première instruction de `f()` qui utilise un des arguments (le premier), nous voyons qu'EBP pointe sur la structure de pile locale, qui est entourée par un rectangle rouge.

Le premier élément de la structure de pile locale est la valeur sauvegardée de EBP, le second est RA, le troisième est le premier argument de la fonction, puis le second et le troisième.

Pour accéder au premier argument de la fonction, on doit ajouter exactement 8 (2 mots de 32-bit) à EBP.

OllyDbg est au courant de cela, c'est pourquoi il a ajouté des commentaires aux éléments de la pile comme

«RETURN from» et «Arg1 = ...», etc.

N.B.: Les arguments de la fonction ne font pas partie de la structure de pile de la fonction, ils font plutôt partie de celle de la fonction appelante.

Par conséquent, OllyDbg a marqué les éléments comme appartenant à une autre structure de pile.

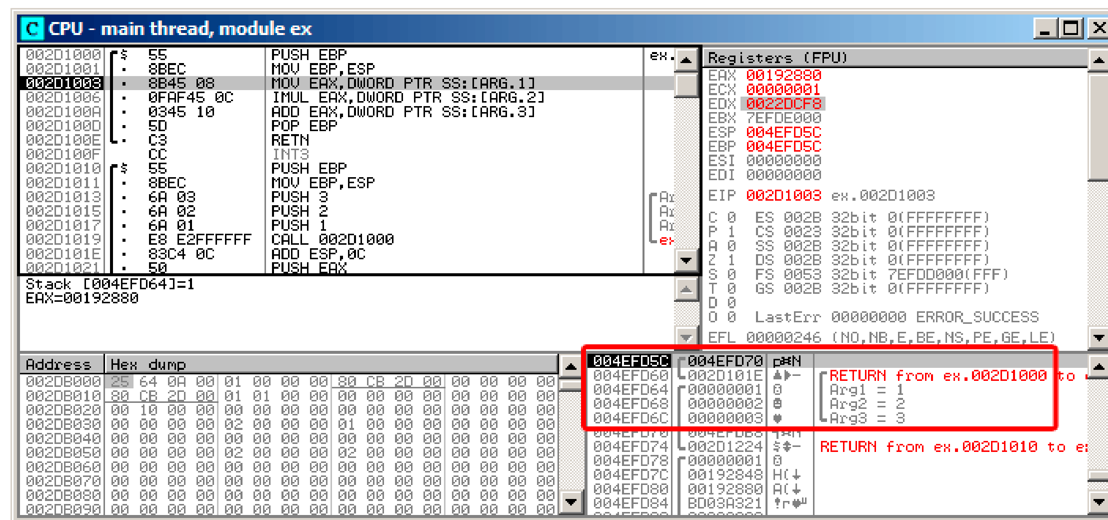


Fig. 1.23: OllyDbg : à l'intérieur de la fonction f()

GCC

Compilons le même code avec GCC 4.4.1 et regardons le résultat dans IDA :

Listing 1.91 : GCC 4.4.1

```
        public f
f       proc near

arg_0   = dword ptr  8
arg_4   = dword ptr  0Ch
arg_8   = dword ptr  10h

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_0] ; 1er argument
        imul    eax, [ebp+arg_4] ; 2ème argument
        add     eax, [ebp+arg_8] ; 3ème argument
        pop     ebp
        retn
f       endp

        public main
main    proc near
```

```
var_10  = dword ptr -10h
var_C   = dword ptr -0Ch
var_8   = dword ptr -8

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 10h
        mov     [esp+10h+var_8], 3 ; 3ème argument
        mov     [esp+10h+var_C], 2 ; 2ème argument
        mov     [esp+10h+var_10], 1 ; 1er argument
        call    f
        mov     edx, offset aD  ; "%d\n"
        mov     [esp+10h+var_C], eax
        mov     [esp+10h+var_10], edx
        call    _printf
        mov     eax, 0
        leave
        retn
main    endp
```

Le résultat est presque le même, avec quelques différences mineures discutées précédemment.

Le pointeur de pile n'est pas remis après les deux appels de fonction (f et printf), car la pénultième instruction LEAVE (.1.6 on page 1332) s'en occupe à la fin.

1.14.2 x64

Le scénario est un peu différent en x86-64. Les arguments de la fonction (les 4 ou 6 premiers d'entre eux) sont passés dans des registres i.e. l'appelée les lit depuis des registres au lieu de les lire dans la pile.

MSVC

MSVC avec optimisation :

Listing 1.92 : MSVC 2012 x64 avec optimisation

```
$SG2997 DB      '%d', 0aH, 00H

main    PROC
        sub     rsp, 40
        mov     edx, 2
        lea     r8d, QWORD PTR [rdx+1] ; R8D=3
        lea     ecx, QWORD PTR [rdx-1] ; ECX=1
        call    f
        lea     rcx, OFFSET FLAT:$SG2997 ; '%d'
        mov     edx, eax
        call    printf
        xor     eax, eax
        add     rsp, 40
```

```
        ret     0
main    ENDP

f       PROC
        ; ECX - 1er argument
        ; EDX - 2ème argument
        ; R8D - 3ème argument
        imul    ecx, edx
        lea     eax, DWORD PTR [r8+rcx]
        ret     0
f       ENDP
```

Comme on peut le voir, la fonction compacte `f()` prend tous ses arguments dans des registres.

La fonction LEA est utilisée ici pour l'addition, apparemment le compilateur considère qu'elle plus rapide que `ADD`.

LEA est aussi utilisée dans la fonction `main()` pour préparer le premier et le troisième argument de `f()`. Le compilateur doit avoir décidé que cela s'exécutera plus vite que la façon usuelle ce charger des valeurs dans les registres, qui utilise l'instruction `MOV`.

Regardons ce qu'a généré MSVC sans optimisation:

Listing 1.93 : MSVC 2012 x64

```
f               proc near

; shadow space:
arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_10          = dword ptr  18h

                ; ECX - 1er argument
                ; EDX - 2ème argument
                ; R8D - 3ème argument
                mov     [rsp+arg_10], r8d
                mov     [rsp+arg_8], edx
                mov     [rsp+arg_0], ecx
                mov     eax, [rsp+arg_0]
                imul    eax, [rsp+arg_8]
                add     eax, [rsp+arg_10]
                retn
f               endp

main            proc near
                sub     rsp, 28h
                mov     r8d, 3 ; 3rd argument
                mov     edx, 2 ; 2nd argument
                mov     ecx, 1 ; 1st argument
                call    f
                mov     edx, eax
                lea     rcx, $SG2931    ; "%d\n"
                call    printf
```

```
                ; retourner 0
                xor     eax, eax
                add     rsp, 28h
                retn
main            endp
```

C'est un peu déroutant, car les 3 arguments dans des registres sont sauvegardés sur la pile pour une certaine raison. Ceci est appelé «shadow space» [82] : chaque Win64 peut (mais ce n'est pas requis) y sauver les 4 registres. Ceci est fait pour deux raisons: 1) c'est trop généreux d'allouer un registre complet (et même 4 registres) pour un argument en entrée, donc il sera accédé par la pile; 2) le debugger sait toujours où trouver les arguments de la fonction lors d'un arrêt [83].

Donc, de grosses fonctions peuvent sauvegarder leurs arguments en entrée dans le «shadow space» si elle veulent les utiliser pendant l'exécution, mais quelques petites fonctions (comme la notre) peuvent ne pas le faire.

C'est la responsabilité de l'appelant d'allouer le «shadow space» sur la pile.

GCC

GCC avec optimisation génère du code plus ou moins compréhensible:

Listing 1.94 : GCC 4.4.6 x64 avec optimisation

```
f:
        ; EDI - 1er argument
        ; ESI - 2ème argument
        ; EDX - 3ème argument
        imul    esi, edi
        lea     eax, [rdx+rsi]
        ret

main:
        sub     rsp, 8
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edi, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, eax
        xor     eax, eax  ; nombre de registres vectoriel passés
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

GCC sans optimisation :

Listing 1.95 : GCC 4.4.6 x64

[82] MSDN
[83] MSDN

```
f:
        ; EDI - 1er argument
        ; ESI - 2ème argument
        ; EDX - 3ème argument
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-8]
        add     eax, DWORD PTR [rbp-12]
        leave
        ret
main:
        push    rbp
        mov     rbp, rsp
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edx, eax
        mov     eax, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, edx
        mov     rdi, rax
        mov     eax, 0  ; nombre de registres vectoriel passés
        call    printf
        mov     eax, 0
        leave
        ret
```

Il n'y a pas d'exigeance de «shadow space» en System V *NIX ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [84]), mais l'appelée peut vouloir sauvegarder ses arguments quelque part en cas de manque de registres.

GCC: uint64_t au lieu de int

Notre exemple fonctionne avec des *int* 32-bit, c'est pourquoi c'est la partie 32-bit des registres qui est utilisée (préfixée par E-).

Il peut être légèrement modifié pour utiliser des valeurs 64-bit:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
        return a*b+c;
};
```

[84] Aussi disponible en https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

```
int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                                0x1111111122222222,
                                0x3333333344444444));
        return 0;
};
```

Listing 1.96 : GCC 4.4.6 x64 avec optimisation

```
f       proc near
        imul    rsi, rdi
        lea     rax, [rdx+rsi]
        retn
f       endp

main    proc near
        sub     rsp, 8
        mov     rdx, 3333333344444444h ; 3ème argument
        mov     rsi, 1111111122222222h ; 2ème argument
        mov     rdi, 1122334455667788h ; 1er argument
        call    f
        mov     edi, offset format ; "%lld\n"
        mov     rsi, rax
        xor     eax, eax ; nombre de registres vectoriel passés
        call    _printf
        xor     eax, eax
        add     rsp, 8
        retn
main    endp
```

Le code est le même, mais cette fois les registres *complets* (préfixés par R-) sont utilisés.

1.14.3 ARM

sans optimisation Keil 6/2013 (Mode ARM)

```
.text:000000A4 00 30 A0 E1      MOV     R3, R0
.text:000000A8 93 21 20 E0      MLA     R0, R3, R1, R2
.text:000000AC 1E FF 2F E1      BX      LR
...
.text:000000B0             main
.text:000000B0 10 40 2D E9      STMFD   SP!, {R4,LR}
.text:000000B4 03 20 A0 E3      MOV     R2, #3
.text:000000B8 02 10 A0 E3      MOV     R1, #2
.text:000000BC 01 00 A0 E3      MOV     R0, #1
.text:000000C0 F7 FF FF EB      BL      f
.text:000000C4 00 40 A0 E1      MOV     R4, R0
.text:000000C8 04 10 A0 E1      MOV     R1, R4
.text:000000CC 5A 0F 8F E2      ADR     R0, aD_0        ; "%d\n"
```

```
.text:000000D0 E3 18 00 EB      BL      __2printf
.text:000000D4 00 00 A0 E3      MOV     R0, #0
.text:000000D8 10 80 BD E8      LDMFD   SP!, {R4,PC}
```

La fonction `main()` appelle simplement deux autres fonctions, avec trois valeurs passées à la première —(`f()`).

Comme il y déjà été écrit, en ARM les 4 premières valeurs sont en général passées par les 4 premiers registres (R0-R3).

La fonction `f()`, comme il semble, utilise les 3 premiers registres (R0-R2) comme arguments.

L'instruction `MLA` (*Multiply Accumulate*) multiplie ses deux premiers opérandes (R3 et R1), additionne le troisième opérande (R2) au produit et stocke le résultat dans le registre zéro (R0), par lequel, d'après le standard, les fonctions retournent leur résultat.

La multiplication et l'addition en une fois (*Fused multiply–add*) est une instruction très utile. À propos, il n'y avait pas une telle instruction en x86 avant les instructions FMA apparues en SIMD [85].

La toute première instruction `MOV R3, R0`, est, apparemment, redondante (car une seule instruction `MLA` pourrait être utilisée à la place ici). Le compilateur ne l'a pas optimisé, puisqu'il n'y a pas l'option d'optimisation.

L'instruction BX rend le contrôle à l'adresse stockée dans le registre LR et, si nécessaire, change le mode du processeur de Thumb à ARM et vice versa. Ceci peut être nécessaire puisque, comme on peut le voir, la fonction `f()` n'est pas au courant depuis quel sorte de code elle peut être appelée, ARM ou Thumb. Ainsi, si elle est appelée depuis du code Thumb, BX ne va pas seulement retourner le contrôle à la fonction appelante, mais également changer le mode du processeur à Thumb. Ou ne pas changer si la fonction a été appelée depuis du code ARM [*ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)A2.3.2].

avec optimisation Keil 6/2013 (Mode ARM)

```
.text:00000098              f
.text:00000098 91 20 20 E0      MLA     R0, R1, R0, R2
.text:0000009C 1E FF 2F E1      BX      LR
```

Et voilà la fonction `f()` compilée par le compilateur Keil en mode optimisation maximale (`-O3`).

L'instruction `MOV` a été supprimée par l'optimisation (ou réduite) et maintenant `MLA` utilise tout les registres contenant les données en entrée et place ensuite le résultat directement dans `R0`, exactement où la fonction appelante va le lire et l'utiliser.

avec optimisation Keil 6/2013 (Mode Thumb)

[85] Wikipédia

```
.text:0000005E 48 43          MULS    R0, R1
.text:00000060 80 18          ADDS    R0, R0, R2
.text:00000062 70 47          BX      LR
```

L'instruction MLA n'est pas disponible dans le mode Thumb, donc le compilateur génère le code effectuant ces deux opérations (multiplication et addition) séparément.

Tout d'abord, la première instruction MULS multiplie R0 par R1, laissant le résultat dans le registreR0. La seconde instruction (ADDS) ajoute le résultat et R2 laissant le résultat dans le registre R0.

ARM64

GCC (Linaro) 4.9 avec optimisation

Tout ce qu'il y a ici est simple. MADD est juste une instruction qui effectue une multiplication/addition fusionnées (similaire à l'instruction MLA que nous avons déjà vue). Tous les 3 arguments sont passés dans la partie 32-bit de X-registres. Effectivement, le type des arguments est *int* 32-bit. Le résultat est renvoyé dans W0.

Listing 1.97 : GCC (Linaro) 4.9 avec optimisation

```
f:
        madd    w0, w0, w1, w2
        ret

main:
; sauver FP et LR dans la pile locale:
        stp     x29, x30, [sp, -16]!
        mov     w2, 3
        mov     w1, 2
        add     x29, sp, 0
        mov     w0, 1
        bl      f
        mov     w1, w0
        adrp    x0, .LC7
        add     x0, x0, :lo12:.LC7
        bl      printf
; retourner 0
        mov     w0, 0
; restaurer FP et LR
        ldp     x29, x30, [sp], 16
        ret

.LC7:
        .string "%d\n"
```

Étendons le type de toutes les données à 64-bit uint64_t et testons:

```
#include <stdio.h>
```

```
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
        return a*b+c;
};

int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                            0x1111111122222222,
                            0x3333333344444444));
        return 0;
};
```

```
f:
        madd    x0, x0, x1, x2
        ret
main:
        mov     x1, 13396
        adrp    x0, .LC8
        stp     x29, x30, [sp, -16]!
        movk    x1, 0x27d0, lsl 16
        add     x0, x0, :lo12:.LC8
        movk    x1, 0x122, lsl 32
        add     x29, sp, 0
        movk    x1, 0x58be, lsl 48
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret

.LC8:
        .string "%lld\n"
```

La fonction `f()` est la même, seulement les X-registres 64-bit sont utilisés entièrement maintenant. Les valeurs longues sur 64-bit sont chargées dans les registres par partie, c'est également décrit ici: 1.39.3 on page 570.

GCC (Linaro) 4.9 sans optimisation

Le code sans optimisation est plus redondant:

```
f:
        sub     sp, sp, #16
        str     w0, [sp,12]
        str     w1, [sp,8]
        str     w2, [sp,4]
        ldr     w1, [sp,12]
        ldr     w0, [sp,8]
        mul     w1, w1, w0
```

```
        ldr     w0, [sp,4]
        add     w0, w1, w0
        add     sp, sp, 16
        ret
```

Le code sauve ses arguments en entrée dans la pile locale, dans le cas où quelqu'un (ou quelque chose) dans cette fonction aurait besoin d'utiliser les registres W0...W2. Cela évite d'écraser les arguments originels de la fonction, qui pourraient être de nouveau utilisés par la suite.

Cela est appelé *Zone de sauvegarde de registre.* [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][86]. L'appelée, toutefois, n'est pas obligée de les sauvegarder. C'est un peu similaire au «Shadow Space» : 1.14.2 on page 137.

Pourquoi est-ce que GCC 4.9 avec l'option d'optimisation supprime ce code de sauvegarde? Parce qu'il a fait plus d'optimisation et en a conclu que les arguments de la fonction n'allaient pas être utilisés par la suite et donc que les registres W0...W2 ne vont pas être utilisés.

Nous avons donc une paire d'instructions MUL/ADD au lieu d'un seul MADD.

1.14.4 MIPS

Listing 1.98 : GCC 4.4.5 avec optimisation

```
.text:00000000 f:
; $a0=a
; $a1=b
; $a2=c
.text:00000000         mult    $a1, $a0
.text:00000004         mflo    $v0
.text:00000008         jr      $ra
.text:0000000C         addu    $v0, $a2, $v0     ; slot de délai de
    branchement
; au retour le résultat est dans $v0
.text:00000010 main:
.text:00000010
.text:00000010 var_10 = -0x10
.text:00000010 var_4  = -4
.text:00000010
.text:00000010         lui     $gp, (__gnu_local_gp >> 16)
.text:00000014         addiu   $sp, -0x20
.text:00000018         la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000001C         sw      $ra, 0x20+var_4($sp)
.text:00000020         sw      $gp, 0x20+var_10($sp)
; définir c:
.text:00000024         li      $a2, 3
; définir a:
.text:00000028         li      $a0, 1
.text:0000002C         jal     f
; définir b:
```

[86]Aussi disponible en http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

```
.text:00000030          li      $a1, 2           ; slot de délai de
    branchement
; le résultat est maintenant dans $v0
.text:00000034          lw      $gp, 0x20+var_10($sp)
.text:00000038          lui     $a0, ($LC0 >> 16)
.text:0000003C          lw      $t9, (printf & 0xFFFF)($gp)
.text:00000040          la      $a0, ($LC0 & 0xFFFF)
.text:00000044          jalr    $t9
; prend le résultat de la fonction f() et le passe
; en second argument à printf():
.text:00000048          move    $a1, $v0         ; slot de délai de
    branchement
.text:0000004C          lw      $ra, 0x20+var_4($sp)
.text:00000050          move    $v0, $zero
.text:00000054          jr      $ra
.text:00000058          addiu   $sp, 0x20 ; slot de délai de branchement
```

Les quatre premiers arguments de la fonction sont passés par quatre registres préfixés par A-.

Il y a deux registres spéciaux en MIPS: HI et LO qui sont remplis avec le résultat 64-bit de la multiplication lors de l'exécution d'une instruction MULT.

Ces registres sont accessibles seulement en utilisant les instructions MFLO et MFHI. Ici MFLO prend la partie basse du résultat de la multiplication et le stocke dans $V0. Donc la partie haute du résultat de la multiplication est abandonnée (le contenu du registre HI n'est pas utilisé). Effectivement: nous travaillons avec des types de données *int* 32-bit ici.

Enfin, ADDU («Add Unsigned » addition non signée) ajoute la valeur du troisième argument au résultat.

Il y a deux instructions différentes pour l'addition en MIPS: ADD et ADDU. La différence entre les deux n'est pas relative au fait d'être signé, mais aux exceptions. ADD peut déclencher une exception lors d'un débordement, ce qui est parfois utile[87] et supporté en ADA LP, par exemple. ADDU ne déclenche pas d'exception lors d'un débordement. Comme C/C++ ne supporte pas ceci, dans notre exemple nous voyons ADDU au lieu de ADD.

Le résultat 32-bit est laissé dans $V0.

Il y a une instruction nouvelle pour nous dans main() : JAL («Jump and Link »).

La différence entre JAL et JALR est qu'un offset relatif est encodé dans la première instruction, tandis que JALR saute à l'adresse absolue stockée dans un registre («Jump and Link Register »).

Les deux fonctions f() et main() sont stockées dans le même fichier objet, donc l'adresse relative de f() est connue et fixée.

[87] http://blog.regehr.org/archives/1154

1.15 Plus loin sur le renvoi des résultats

En x86, le résultat de l'exécution d'une fonction est d'habitude renvoyé [88] dans le registre EAX.

Si il est de type octet ou un caractère (*char*), alors la partie basse du registre EAX (AL) est utilisée. Si une fonction renvoie un nombre de type *float*, le registre ST(0) du FPU est utilisé. En ARM, d'habitude, le résultat est renvoyé dans le registre R0.

1.15.1 Tentative d'utilisation du résultat d'une fonction renvoyant *void*

Donc, que se passe-t-il si le type de retour de la fonction main() a été déclaré du type *void* et non pas *int*? Ce que l'on nomme le code de démarrage (startup-code) appelle main() grosso-modo de la façon suivante:

```
push envp
push argv
push argc
call main
push eax
call exit
```

En d'autres mots:

```
exit(main(argc,argv,envp));
```

Si vous déclarez main() comme renvoyant *void*, rien ne sera renvoyé explicitement (en utilisant la déclaration *return*), alors quelque chose d'inconnu, qui aura été stocké dans la registre EAX lors de l'exécution de main() sera l'unique argument de la fonction exit(). Il y aura probablement une valeur aléatoire, laissée lors de l'exécution de la fonction, donc le code de retour du programme est pseudo-aléatoire.

Illustrons ce fait: Notez bien que la fonction main() a un type de retour *void* :

```
#include <stdio.h>

void main()
{
        printf ("Hello, world!\n");
};
```

Compilons-le sous Linux.

GCC 4.8.1 a remplacé printf() par puts() (nous avons vu ceci avant: 1.5.3 on page 29), mais c'est OK, puisque puts() renvoie le nombre de caractères écrit, tout comme printf(). Remarquez que le registre EAX n'est pas mis à zéro avant la fin de main().

Ceci implique que la valeur de EAX à la fin de main() contient ce que puts() y avait mis.

[88] Voir également : MSDN: Return Values (C++) : MSDN

Listing 1.99 : GCC 4.8.1

```
.LC0:
        .string "Hello, world!"
main:
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0
        call    puts
        leave
        ret
```

Écrivons un script bash affichant le code de retour:

Listing 1.100 : tst.sh

```
#!/bin/sh
./hello_world
echo $?
```

Et lançons le:

```
$ tst.sh
Hello, world!
14
```

14 est le nombre de caractères écrits. Le nombre de caractères affichés est *passé* de `printf()`, à travers EAX/RAX, dans le «code de retour».

Un autre exemple dans le livre: 3.32 on page 833.

À propos, lorsque l'on décompile du C++ dans Hex-Rays, nous rencontrons souvent une fonction qui se termine par un destructeur d'une classe:

```
...

call    ??1CString@@QAE@XZ ; CString:: CString(void)
mov     ecx, [esp+30h+var_C]
pop     edi
pop     ebx
mov     large fs:0, ecx
add     esp, 28h
retn
```

Dans le standard C++, le destructeur ne renvoie rien, mais lorsque Hex-Rays n'en sait rien, et pense que le destructeur et cette fonction renvoient tout deux un *int*

```
...

        return CString::~CString(&Str);
}
```

1.15.2 Que se passe-t-il si on n'utilise pas le résultat de la fonction?

`printf()` renvoie le nombre de caractères écrit avec succès, mais, en pratique, ce résultat est rarement utilisé.

Il est aussi possible d'appeler une fonction dont la finalité est de renvoyer une valeur, et de ne pas l'utiliser:

```
int f()
{
    // skip first 3 random values:
    rand();
    rand();
    rand();
    // and use 4th:
    return rand();
};
```

Le résultat de la fonction rand() est mis dans EAX, dans les quatre cas.

Mais dans les 3 premiers, la valeur dans EAX n'est pas utilisée.

1.15.3 Renvoyer une structure

Revenons au fait que la valeur de retour est passée par le registre EAX.

C'est pourquoi les vieux compilateurs C ne peuvent pas créer de fonction capable de renvoyer quelque chose qui ne tient pas dans un registre (d'habitude *int*), mais si besoin, les informations doivent être renvoyées via un pointeur passé en argument.

Donc, d'habitude, si une fonction doit renvoyer plusieurs valeurs, elle en renvoie une seule, et le reste—par des pointeurs.

Maintenant, il est possible de renvoyer, disons, une structure entière, mais ce n'est pas encore très populaire. Si une fonction doit renvoyer une grosse structure, la fonction appelante doit l'allouer et passer un pointeur sur cette dernière via le premier argument, de manière transparente pour le programmeur. C'est presque la même chose que de passer un pointeur manuellement dans le premier argument, mais le compilateur le cache.

Petit exemple:

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    struct s rt;
```

```
    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;

    return rt;
};
```

…ce que nous obtenons (MSVC 2010 /Ox) :

```
$T3853 = 8                        ; size = 4
_a$ = 12                          ; size = 4
?get_some_values@@YA?AUs@@H@Z PROC            ; get_some_values
    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, DWORD PTR $T3853[esp-4]
    lea    edx, DWORD PTR [ecx+1]
    mov    DWORD PTR [eax], edx
    lea    edx, DWORD PTR [ecx+2]
    add    ecx, 3
    mov    DWORD PTR [eax+4], edx
    mov    DWORD PTR [eax+8], ecx
    ret    0
?get_some_values@@YA?AUs@@H@Z ENDP            ; get_some_values
```

Ici, le nom de la macro interne pour passer le pointeur sur une structure est $T3853.

Cet exemple peut être récrit en utilisant les extensions C99 du langage:

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    return (struct s){.a=a+1, .b=a+2, .c=a+3};
};
```

Listing 1.101 : GCC 4.8.1

```
_get_some_values proc near

ptr_to_struct   = dword ptr  4
a               = dword ptr  8

                mov     edx, [esp+a]
                mov     eax, [esp+ptr_to_struct]
                lea     ecx, [edx+1]
                mov     [eax], ecx
                lea     ecx, [edx+2]
                add     edx, 3
                mov     [eax+4], ecx
                mov     [eax+8], edx
```

```
                retn
_get_some_values endp
```

Comme on le voit, la fonction remplit simplement les champs de la structure allouée par la fonction appelante, comme si un pointeur sur la structure avait été passé. Donc, il n'y a pas d'impact négatif sur les performances.

1.16 Pointeurs

1.16.1 Renvoyer des valeurs

Les pointeurs sont souvent utilisés pour renvoyer des valeurs depuis les fonctions (rappelez-vous le cas (1.12 on page 91) de `scanf()`).

Par exemple, lorsqu'une fonction doit renvoyer deux valeurs.

Exemple avec des variables globales

```
#include <stdio.h>

void f1 (int x, int y, int *sum, int *product)
{
        *sum=x+y;
        *product=x*y;
};

int sum, product;

void main()
{
        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

Ceci se compile en:

Listing 1.102 : MSVC 2010 avec optimisation (/Ob0)

```
COMM    _product:DWORD
COMM    _sum:DWORD
$SG2803 DB      'sum=%d, product=%d', 0aH, 00H

_x$ = 8                 ; size = 4
_y$ = 12                ; size = 4
_sum$ = 16              ; size = 4
_product$ = 20          ; size = 4
_f1     PROC
        mov     ecx, DWORD PTR _y$[esp-4]
        mov     eax, DWORD PTR _x$[esp-4]
        lea     edx, DWORD PTR [eax+ecx]
        imul    eax, ecx
```

```
        mov     ecx, DWORD PTR _product$[esp-4]
        push    esi
        mov     esi, DWORD PTR _sum$[esp]
        mov     DWORD PTR [esi], edx
        mov     DWORD PTR [ecx], eax
        pop     esi
        ret     0
_f1     ENDP

_main   PROC
        push    OFFSET _product
        push    OFFSET _sum
        push    456     ; 000001c8H
        push    123     ; 0000007bH
        call    _f1
        mov     eax, DWORD PTR _product
        mov     ecx, DWORD PTR _sum
        push    eax
        push    ecx
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
        add     esp, 28
        xor     eax, eax
        ret     0
_main   ENDP
```

Regardons ceci dans OllyDbg :

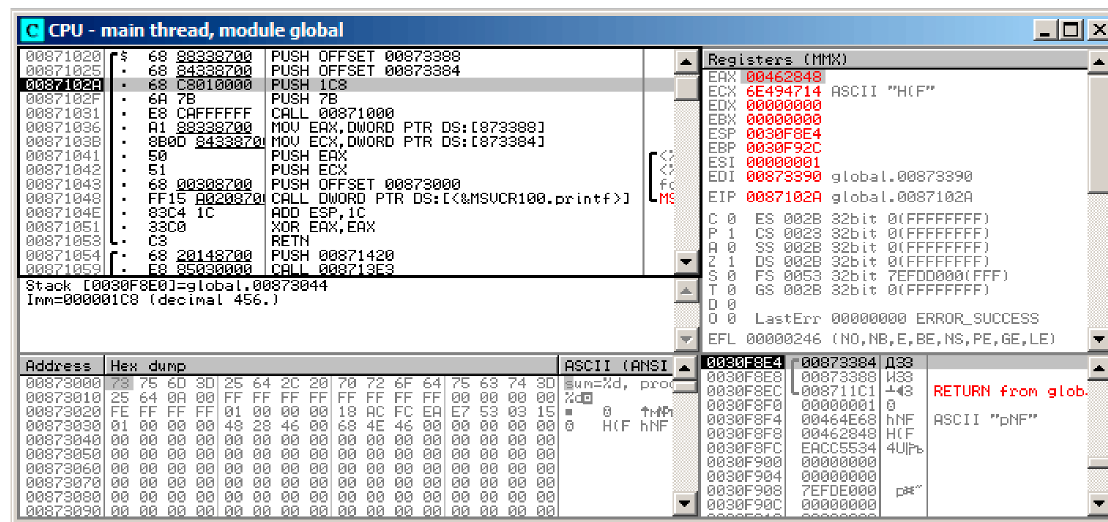


Fig. 1.24: OllyDbg : les adresses des variables globales sont passées à `f1()`

Tout d'abord, les adresses des variables globales sont passées à `f1()`. Nous pouvons cliquer sur «Follow in dump» sur l'élément de la pile, et nous voyons l'espace alloué dans le segment de données pour les deux variables.

Ces variables sont mises à zéro, car les données non-initialisées (de BSS) sont effacées avant le début de l'exécution, [voir *ISO/IEC 9899:TC3 (C C99 standard)*, (2007) 6.7.8p10].

Elles se trouvent dans le segment de données, nous pouvons le vérifier en appuyant sur Alt-M et en regardant la carte de la mémoire:

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00050000	00004000				Map	R	R	
00060000	00001000				Priv	RW	RW	
00070000	00067000				Map	R	R	C:\Windows\System32\loc
00159000	00007000				Priv	RW Gua:	RW Gua:	
0030D000	00001000				Priv	RW Gua:	RW Gua:	
0030E000	00002000			Stack of main thread	Priv	RW	RW	
00460000	00005000			Heap	Priv	RW	RW	
004A0000	00007000				Priv	RW	RW	
006B0000	0000C000			Default heap	Priv	RW	RW	
00870000	00001000	global		PE header	Img	R	RWE Cop	
00871000	00001000	global	.text	Code	Img	R E	RWE Cop	
00872000	00001000	global	.rdata	Imports	Img	R	RWE Cop	
00873000	00001000	global	.data	Data	Img	RW	RWE Cop	
00874000	00001000	global	.reloc	Relocations	Img	R	RWE Cop	
6E3E0000	00001000	MSVCR100		PE header	Img	R	RWE Cop	
6E3E1000	000B2000	MSVCR100	.text	Code, imports, exports	Img	R E	RWE Cop	
6E493000	00006000	MSVCR100	.data	Data	Img	RW Cop	RWE Cop	
6E499000	00001000	MSVCR100	.rsrc	Resources	Img	R	RWE Cop	
6E49A000	00005000	MSVCR100	.reloc	Relocations	Img	R	RWE Cop	
755D0000	00001000	Mod_755D		PE header	Img	R	RWE Cop	
755D1000	00003000				Img	R E	RWE Cop	
755D4000	00001000				Img	RW	RWE Cop	
755D5000	00003000				Img	R	RWE Cop	
755E0000	00001000	Mod_755E		PE header	Img	R	RWE Cop	
755E1000	0004D000				Img	R E	RWE Cop	
7562E000	00005000				Img	RW Cop	RWE Cop	
75633000	00009000				Img	R	RWE Cop	

Fig. 1.25: OllyDbg : carte de la mémoire

Traçons l'exécution (F7) jusqu'au début de `f1()` :

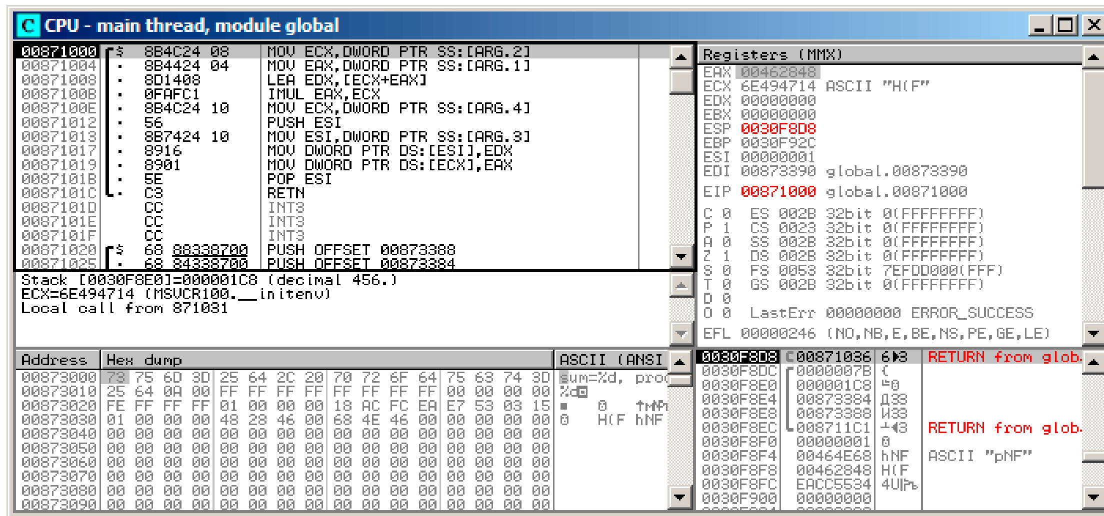


Fig. 1.26: OllyDbg : `f1()` commence

Deux valeurs sont visibles sur la pile: 456 (`0x1C8`) et 123 (`0x7B`), et aussi les adresses des deux variables globales.

Suivons l'exécution jusqu'à la fin de `f1()`. Dans la fenêtre en bas à gauche, nous voyons comment le résultat du calcul apparaît dans les variables globales:

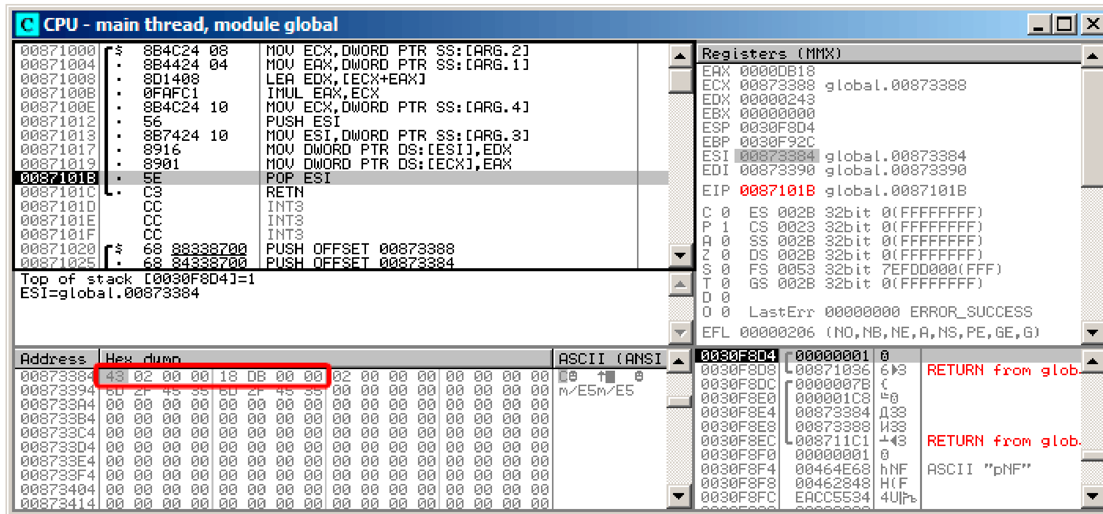


Fig. 1.27: OllyDbg : l'exécution de `f1()` est terminée

Maintenant les valeurs des variables globales sont chargées dans des registres, prêtes à être passées à `printf()` (via la pile) :

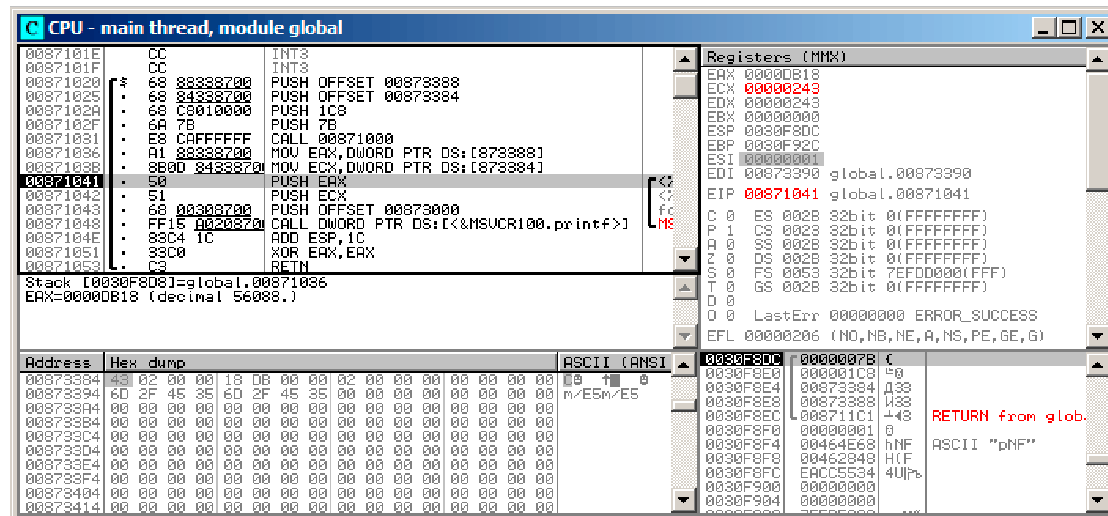


Fig. 1.28: OllyDbg : les adresses des variables globales sont passées à `printf()`

Exemple avec des variables locales

Modifions légèrement notre exemple:

Listing 1.103 : maintenant les variables sum et product sont locales

```
void main()
{
        int sum, product; // maintenant, les variables sont locales à la
    fonction

        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

Le code de `f1()` ne va pas changer. Seul celui de `main()` va changer:

Listing 1.104 : MSVC 2010 avec optimisation (/Ob0)

```
_product$ = -8          ; size = 4
_sum$ = -4              ; size = 4
_main   PROC
; Line 10
        sub     esp, 8
; Line 13
        lea     eax, DWORD PTR _product$[esp+8]
        push    eax
        lea     ecx, DWORD PTR _sum$[esp+12]
        push    ecx
```

```
        push    456     ; 000001c8H
        push    123     ; 0000007bH
        call    _f1
; Line 14
        mov     edx, DWORD PTR _product$[esp+24]
        mov     eax, DWORD PTR _sum$[esp+24]
        push    edx
        push    eax
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
; Line 15
        xor     eax, eax
        add     esp, 36
        ret     0
```

Regardons à nouveau avec OllyDbg. Les adresses des variables locales dans la pile sont 0x2EF854 et 0x2EF858. Voyons comment elles sont poussées sur la pile:

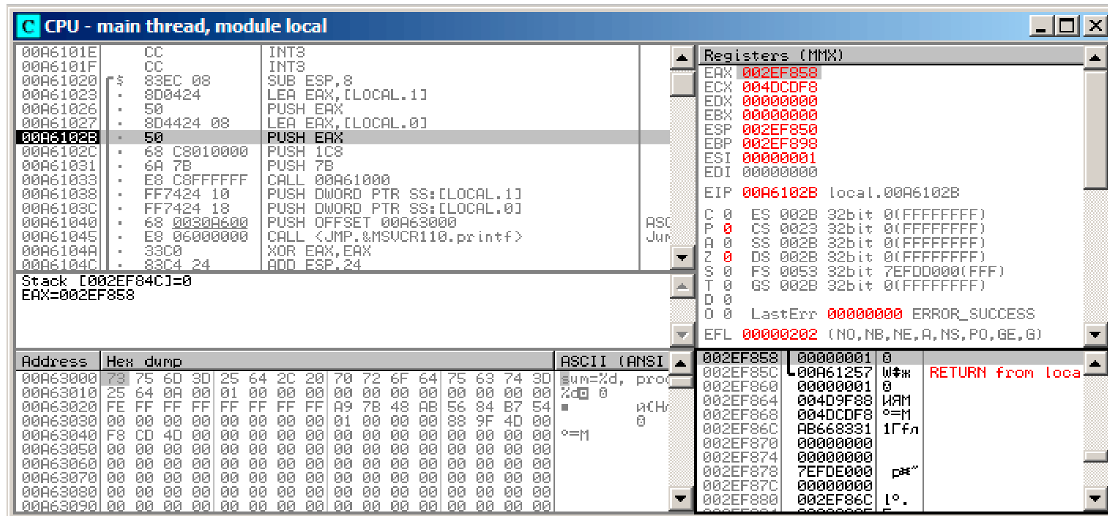


Fig. 1.29: OllyDbg : les adresses des variables locales sont poussées sur la pile

f1() commence. Jusqu'ici, il n'y a que des restes de données sur la pile en 0x2EF854 et 0x2EF858 :

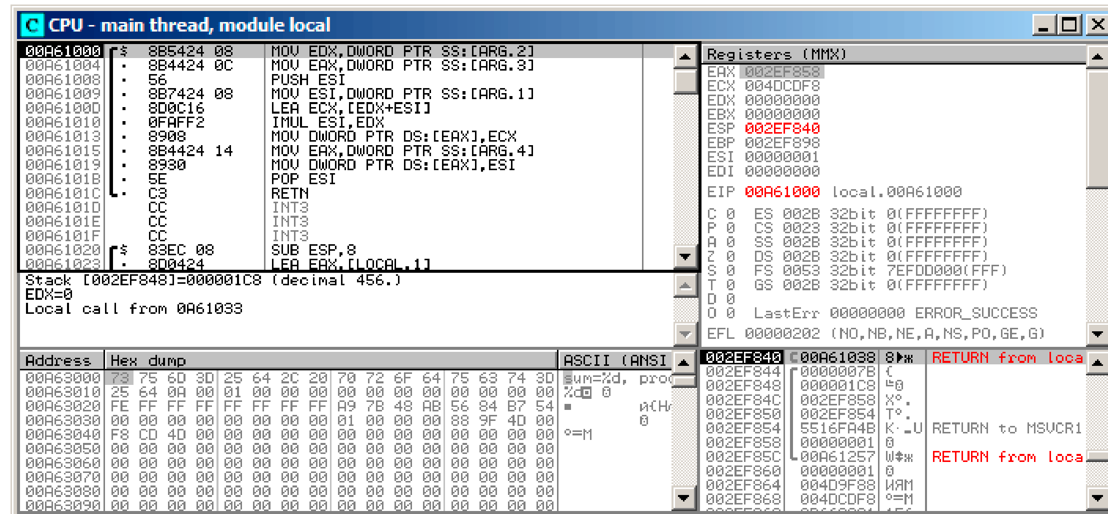


Fig. 1.30: OllyDbg : f1() commence

f1() se termine:

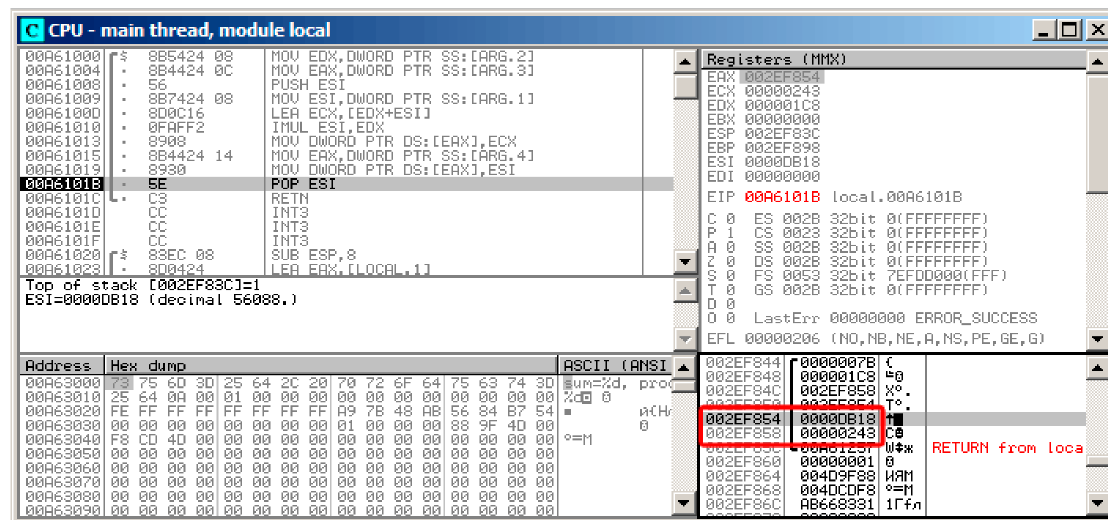


Fig. 1.31: OllyDbg : f1() termine son exécution

Nous trouvons maintenant 0xDB18 et 0x243 aux adresses 0x2EF854 et 0x2EF858. Ces valeurs sont les résultats de f1().

Conclusion

f1() peut renvoyer des pointeurs sur n'importe quel emplacement en mémoire, situés n'importe où.

C'est par essence l'utilité des pointeurs.

À propos, les *references* C++ fonctionnent exactement pareil. Voir à ce propos: (3.21.3 on page 721).

1.16.2 Échanger les valeurs en entrée

Ceci fait ce que l'on veut:

```
#include <memory.h>
#include <stdio.h>

void swap_bytes (unsigned char* first, unsigned char* second)
{
        unsigned char tmp1;
        unsigned char tmp2;

        tmp1=*first;
        tmp2=*second;
```

```
        *first=tmp2;
        *second=tmp1;
};

int main()
{
        // copier la chaîne dans la heap, afin de pouvoir la modifier
        char *s=strdup("string");

        // échanger le 2ème et le 3ème caractères
        swap_bytes (s+1, s+2);

        printf ("%s\n", s);
};
```

Comme on le voit, les octets sont chargés dans la partie 8-bit basse de ECX et EBX en utilisant MOVZX (donc les parties hautes de ces registres vont être effacées) et ensuite les octets échangés sont récrits.

Listing 1.105 : GCC 5.4 avec optimisation

```
swap_bytes:
        push    ebx
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+12]
        movzx   ecx, BYTE PTR [edx]
        movzx   ebx, BYTE PTR [eax]
        mov     BYTE PTR [edx], bl
        mov     BYTE PTR [eax], cl
        pop     ebx
        ret
```

Les adresses des deux octets sont lues depuis les arguments et durant l'exécution de la fonction sont copiés dans EDX et EAX.

Donc nous utilisons des pointeurs, il n'y a sans doute pas de meilleure façon de réaliser cette tâche sans eux.

1.17 Opérateur GOTO

L'opérateur GOTO est en général considéré comme un anti-pattern, voir [Edgar Dijkstra, *Go To Statement Considered Harmful* (1968)[89]]. Néanmoins, il peut être utilisé raisonnablement, voir [Donald E. Knuth, *Structured Programming with go to Statements* (1974)[90]] [91].

Voici un exemple très simple:

```
#include <stdio.h>

```

[89] http://yurichev.com/mirrors/Dijkstra68.pdf
[90] http://yurichev.com/mirrors/KnuthStructuredProgrammingGoTo.pdf
[91] [Dennis Yurichev, *C/C++ programming language notes*] a aussi quelques exemples.

```
int main()
{
        printf ("begin\n");
        goto exit;
        printf ("skip me!\n");
exit:
        printf ("end\n");
};
```

Voici ce que nous obtenons avec MSVC 2012:

Listing 1.106 : MSVC 2012

```
$SG2934 DB      'begin', 0aH, 00H
$SG2936 DB      'skip me!', 0aH, 00H
$SG2937 DB      'end', 0aH, 00H

_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2934 ; 'begin'
        call    _printf
        add     esp, 4
        jmp     SHORT $exit$3
        push    OFFSET $SG2936 ; 'skip me!'
        call    _printf
        add     esp, 4
$exit$3:
        push    OFFSET $SG2937 ; 'end'
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

L'instruction *goto* a simplement été remplacée par une instruction JMP, qui a le même effet: un saut inconditionnel à un autre endroit. Le second printf() peut seulement être exécuté avec une intervention humaine, en utilisant un débogueur ou en modifiant le code.

Cela peut être utile comme exercice simple de patching. Ouvrons l'exécutable généré dans Hiew:

```
Hiew: goto.exe
   C:\Polygon\goto.exe                          □FRO --------        a32 PE .00401000
.00401000: 55                                   push         ebp
.00401001: 8BEC                                 mov          ebp,esp
.00401003: 6800304000                           push         000403000 ;'begin' --□1
.00401008: FF1590204000                         call         printf
.0040100E: 83C404                               add          esp,4
.00401011: EB0E                                 jmps        .000401021 --□2
.00401013: 6808304000                           push         000403008 ;'skip me!' --□3
.00401018: FF1590204000                         call         printf
.0040101E: 83C404                               add          esp,4
.00401021: 6814304000                          2push         000403014 --□4
.00401026: FF1590204000                         call         printf
.0040102C: 83C404                               add          esp,4
.0040102F: 33C0                                 xor          eax,eax
.00401031: 5D                                   pop          ebp
.00401032: C3                                   retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
```

Fig. 1.32: Hiew

Placez le curseur à l'adresse du JMP (0x410), pressez F3 (edit), pressez deux fois zéro, donc l'opcode devient EB 00 :

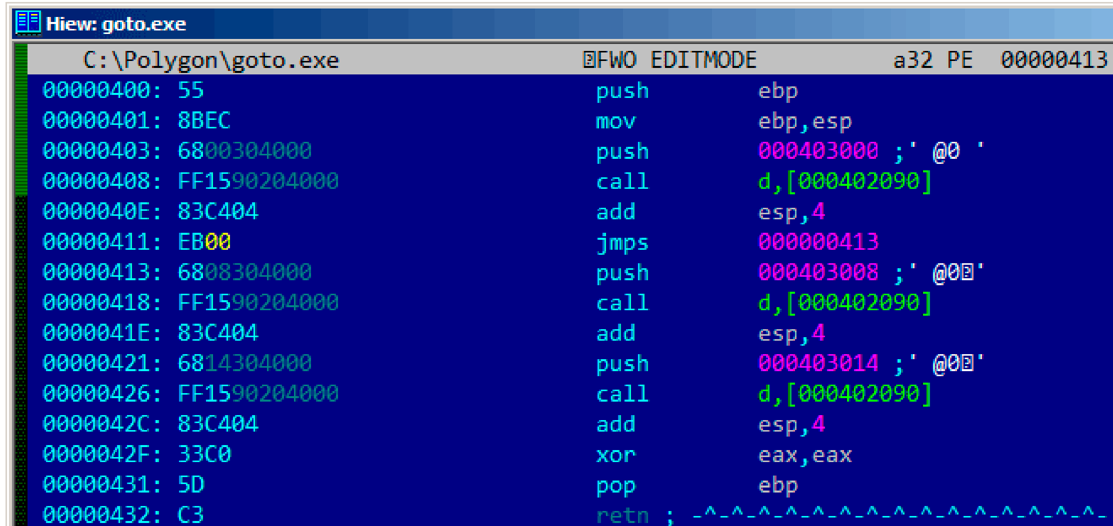


Fig. 1.33: Hiew

Le second octet de l'opcode de JMP indique l'offset relatif du saut, 0 signifie le point juste après l'instruction courante.

Donc maintenant JMP n'évite plus le second printf().

Pressez F9 (save) et quittez. Maintenant, si nous lançons l'exécutable, nous verrons ceci:

Listing 1.107 : Sortie de l'exécutable modifié

```
C:\...>goto.exe

begin
skip me!
end
```

Le même résultat peut être obtenu en remplaçant l'instruction JMP par 2 instructions NOP.

NOP a un opcode de 0x90 et une longueur de 1 octet, donc nous en avons besoin de 2 pour remplacer JMP (qui a une taille de 2 octets).

1.17.1 Code mort

Le second appel à printf() est aussi appelé «code mort» en terme de compilateur.

Cela signifie que le code ne sera jamais exécuté. Donc lorsque vous compilez cet exemple avec les optimisations, le compilateur supprime le «code mort», n’en laissant aucune trace:

Listing 1.108 : MSVC 2012 avec optimisation

```
$SG2981 DB      'begin', 0aH, 00H
$SG2983 DB      'skip me!', 0aH, 00H
$SG2984 DB      'end', 0aH, 00H

_main   PROC
        push    OFFSET $SG2981 ; 'begin'
        call    _printf
        push    OFFSET $SG2984 ; 'end'
$exit$4:
        call    _printf
        add     esp, 8
        xor     eax, eax
        ret     0
_main   ENDP
```

Toutefois, le compilateur a oublié de supprimer la chaîne «skip me!».

1.17.2 Exercice

Essayez d’obtenir le même résultat en utilisant votre compilateur et votre débogueur favoris.

1.18 Sauts conditionnels

1.18.1 Exemple simple

```
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
```

```
        printf ("a<b\n");
};

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
};
```

x86

x86 + MSVC

Voici à quoi ressemble la fonction `f_signed()` :

Listing 1.109 : MSVC 2010 sans optimisation

```
_a$ = 8
_b$ = 12
_f_signed PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jle    SHORT $LN3@f_signed
    push   OFFSET $SG737        ; 'a>b'
    call   _printf
    add    esp, 4
$LN3@f_signed:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_signed
    push   OFFSET $SG739        ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_signed:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jge    SHORT $LN4@f_signed
    push   OFFSET $SG741        ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_signed:
    pop    ebp
    ret    0
_f_signed ENDP
```

La première instruction, JLE, représente *Jump if Less or Equal* (saut si inférieur ou égal). En d'autres mots, si le deuxième opérande est plus grand ou égal au premier, le flux d'exécution est passé à l'adresse ou au label spécifié dans l'instruction. Si la condition ne déclenche pas le saut, car le second opérande est plus petit que le

premier, le flux d'exécution ne sera pas altéré et le premier `printf()` sera exécuté. Le second test est `JNE` : *Jump if Not Equal* (saut si non égal). Le flux d'exécution ne changera pas si les opérandes sont égaux.

Le troisième test est `JGE` : *Jump if Greater or Equal*—saute si le premier opérande est supérieur ou égal au deuxième. Donc, si les trois sauts conditionnels sont effectués, aucun des appels à `printf()` ne sera exécuté. Ceci est impossible sans intervention spéciale. Regardons maintenant la fonction `f_unsigned()`. La fonction `f_unsigned()` est la même que `f_signed()`, à la différence que les instructions `JBE` et `JAE` sont utilisées à la place de `JLE` et `JGE`, comme suit:

Listing 1.110 : GCC

```
_a$ = 8   ; size = 4
_b$ = 12  ; size = 4
_f_unsigned PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jbe    SHORT $LN3@f_unsigned
    push   OFFSET $SG2761    ; 'a>b'
    call   _printf
    add    esp, 4
$LN3@f_unsigned:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_unsigned
    push   OFFSET $SG2763    ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_unsigned:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jae    SHORT $LN4@f_unsigned
    push   OFFSET $SG2765    ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_unsigned:
    pop    ebp
    ret    0
_f_unsigned ENDP
```

Comme déjà mentionné, les instructions de branchement sont différentes: `JBE`—*Jump if Below or Equal* (saut si inférieur ou égal) et `JAE`—*Jump if Above or Equal* (saut si supérieur ou égal). Ces instructions (`JA`/`JAE`/`JB`/`JBE`) diffèrent de `JG`/`JGE`/`JL`/`JLE` par le fait qu'elles travaillent avec des nombres non signés.

C'est pourquoi si nous voyons que `JG`/`JL` sont utilisés à la place de `JA`/`JB` ou vice-versa, nous pouvons être presque sûr que les variables sont signées ou non signées, respectivement. Voici la fonction `main()`, où presque rien n'est nouveau pour nous:

Listing 1.111 : `main()`

```
_main   PROC
        push    ebp
        mov     ebp, esp
        push    2
        push    1
        call    _f_signed
        add     esp, 8
        push    2
        push    1
        call    _f_unsigned
        add     esp, 8
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

x86 + MSVC + OllyDbg

Nous pouvons voir comment les flags sont mis en lançant cet exemple dans OllyDbg. Commençons par `f_unsigned()`, qui utilise des entiers non signés.

CMP est exécuté trois fois ici, mais avec les même arguments, donc les flags sont les même à chaque fois.

Résultat de la première comparaison:

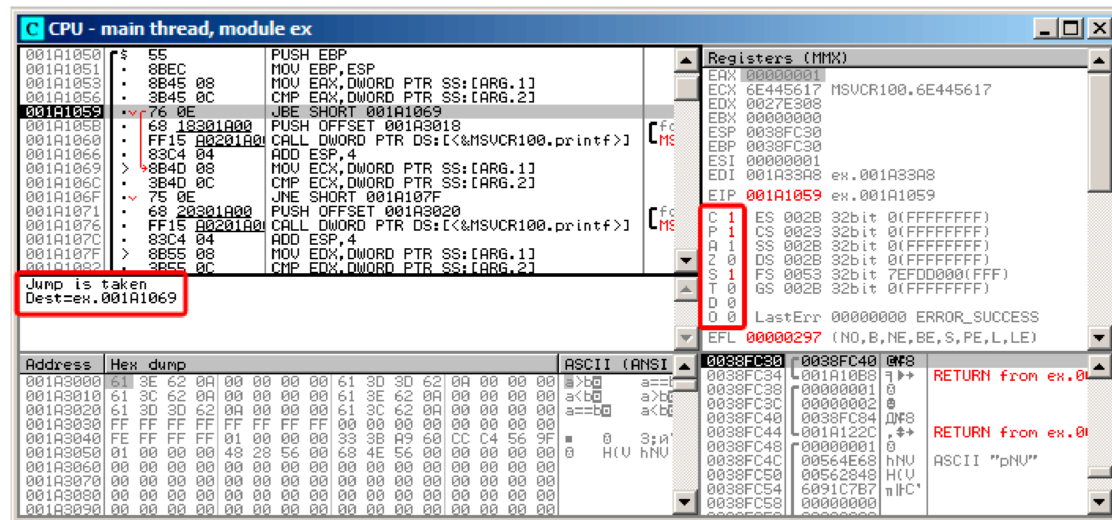


Fig. 1.34: OllyDbg : `f_unsigned()` : premier saut conditionnel

Donc, les flags sont: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0.

Ils sont nommés avec une seule lettre dans OllyDbg par concision.

OllyDbg indique que le saut (JBE) va être suivi maintenant. En effet, si nous regardons dans le manuel d'Intel (12.1.4 on page 1315), nous pouvons lire que JBE est déclenchée si CF=1 ou ZF=1. La condition est vraie ici, donc le saut est effectué.

Le saut conditionnel suivant:

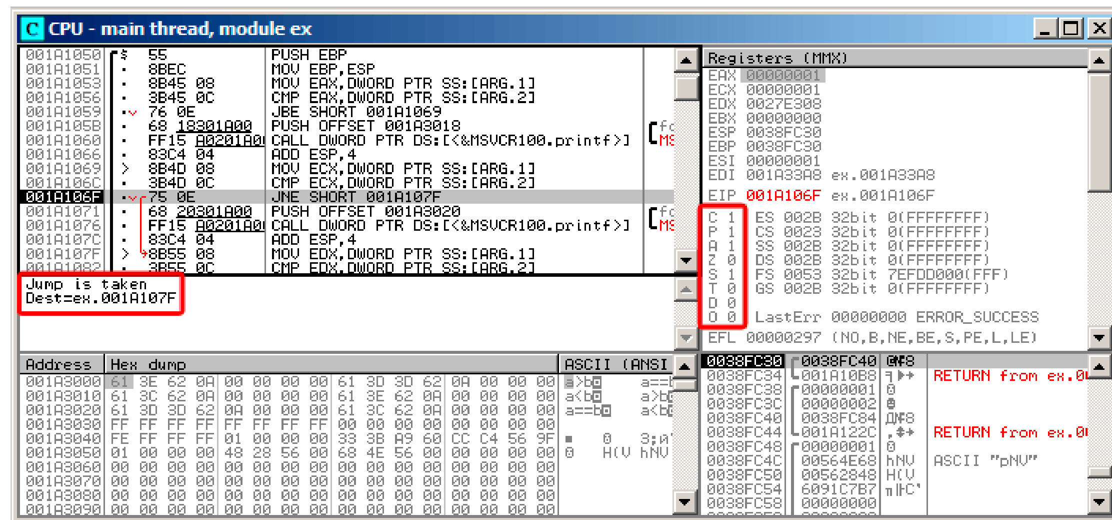


Fig. 1.35: OllyDbg : `f_unsigned()` : second saut conditionnel

OllyDbg indique que le saut JNZ va être effectué maintenant. En effet, JNZ est déclenché si ZF=0 (Zero Flag).

Le troisième saut conditionnel, JNB :

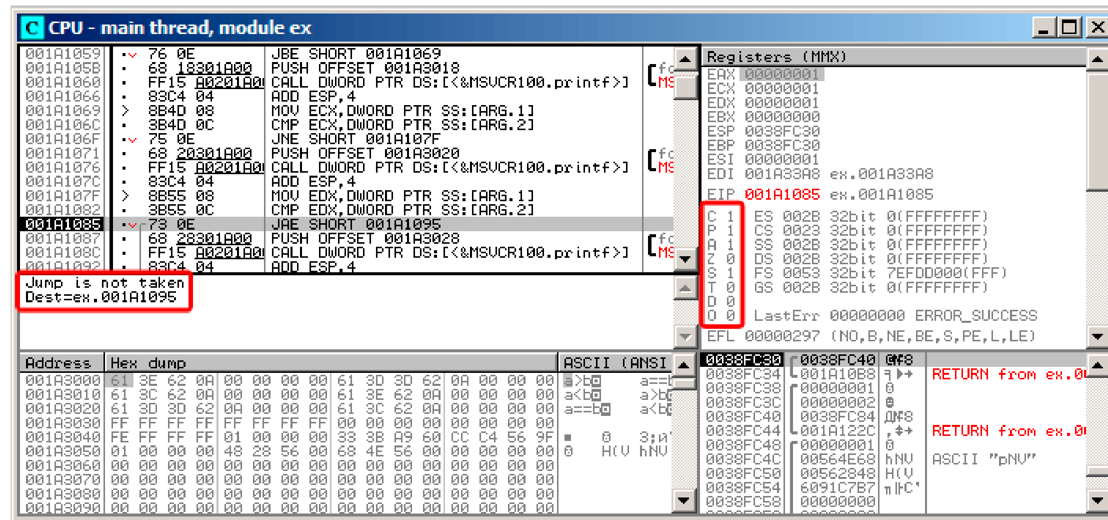


Fig. 1.36: OllyDbg : `f_unsigned()` : troisième saut conditionnel

Dans le manuel d'Intel (12.1.4 on page 1315) nous pouvons voir que JNB est déclenché si CF=0 (Carry Flag - flag de retenue). Ce qui n'est pas vrai dans notre cas, donc le troisième `printf()` sera exécuté.

Maintenant, regardons la fonction `f_signed()`, qui fonctionne avec des entiers non signés. Les flags sont mis de la même manière: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. Le premier saut conditionnel JLE est effectué:

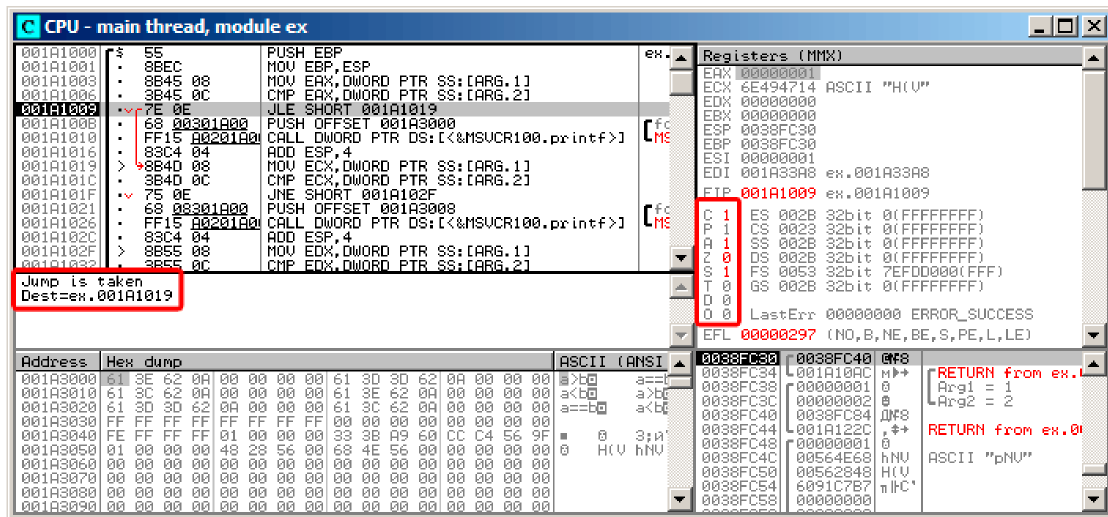


Fig. 1.37: OllyDbg : `f_signed()` : premier saut conditionnel

Dans les manuels d'Intel (12.1.4 on page 1315) nous trouvons que cette instruction est déclenchée si ZF=1 ou SF≠OF. SF≠OF dans notre cas, donc le saut est effectué.

Le second saut conditionnel, JNZ, est effectué: si ZF=0 (Zero Flag) :

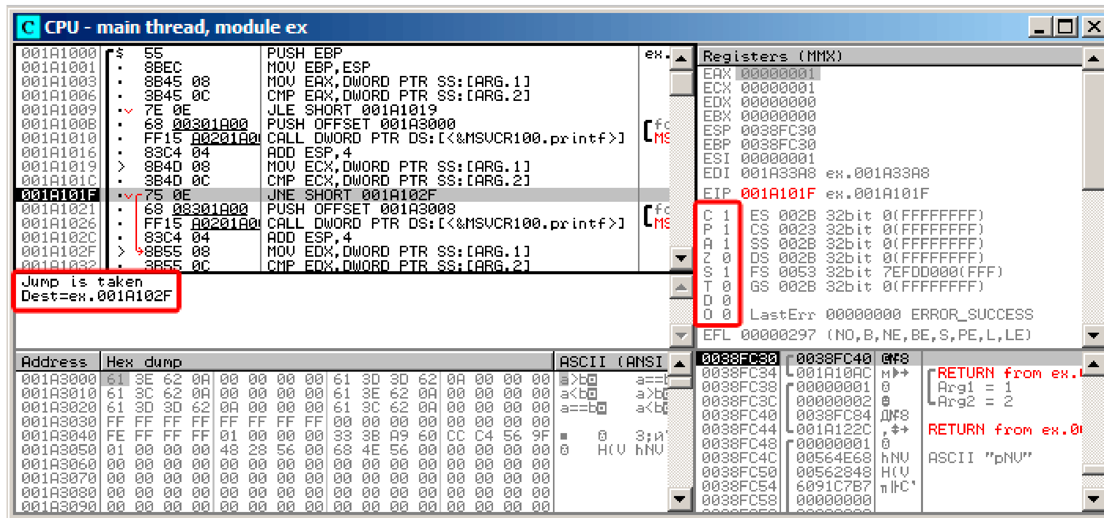


Fig. 1.38: OllyDbg : f_signed() : second saut conditionnel

Le troisième saut conditionnel, JGE, ne sera pas effectué car il ne l'est que si SF=OF, et ce n'est pas vrai dans notre cas:

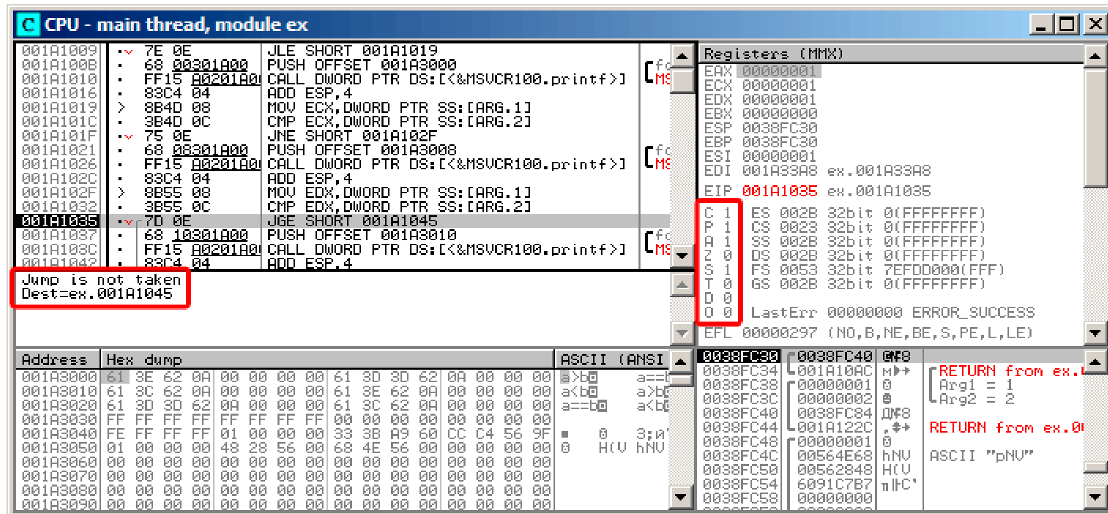


Fig. 1.39: OllyDbg : `f_signed()` : troisième saut conditionnel

x86 + MSVC + Hiew

Nous pouvons essayer de patcher l'exécutable afin que la fonction `f_unsigned()` affiche toujours «a==b», quelque soient les valeurs en entrée. Voici à quoi ça ressemble dans Hiew:

```
Hiew: 7_1.exe
   C:\Polygon\ollydbg\7_1.exe              FRO --------          a32 PE .00401000 Hiew 8.02 (c)SEN
.00401000: 55                           push        ebp
.00401001: 8BEC                         mov         ebp,esp
.00401003: 8B4508                       mov         eax,[ebp][8]
.00401006: 3B450C                       cmp         eax,[ebp][00C]
.00401009: 7E0D                         jle        .000401018 --1
.0040100B: 6800B04000                   push        00040B000 --2
.00401010: E8AA000000                   call       .0004010BF --3
.00401015: 83C404                       add         esp,4
.00401018: 8B4D08                      1mov         ecx,[ebp][8]
.0040101B: 3B4D0C                       cmp         ecx,[ebp][00C]
.0040101E: 750D                         jnz        .00040102D --4
.00401020: 6808B04000                   push        00040B008 ;'a==b' --5
.00401025: E895000000                   call       .0004010BF --3
.0040102A: 83C404                       add         esp,4
.0040102D: 8B5508                      4mov         edx,[ebp][8]
.00401030: 3B550C                       cmp         edx,[ebp][00C]
.00401033: 7D0D                         jge        .000401042 --6
.00401035: 6810B04000                   push        00040B010 --7
.0040103A: E880000000                   call       .0004010BF --3
.0040103F: 83C404                       add         esp,4
.00401042: 5D                          6pop         ebp
.00401043: C3                           retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.00401044: CC                           int         3
.00401045: CC                           int         3
.00401046: CC                           int         3
.00401047: CC                           int         3
.00401048: CC                           int         3
1Global 2FilBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 9 1byte 10Leave 11Naked 12AddNam
```

Fig. 1.40: Hiew: fonction `f_unsigned()`

Essentiellement, nous devons accomplir ces trois choses:

- forcer le premier saut à toujours être effectué;
- forcer le second saut à n'être jamais effectué;
- forcer le troisième saut à toujours être effectué.

Nous devons donc diriger le déroulement du code pour toujours effectuer le second `printf()`, et afficher «a==b».

Trois instructions (ou octets) doivent être modifiées:

- Le premier saut devient un JMP, mais l'offset reste le même.
- Le second saut peut être parfois suivi, mais dans chaque cas il sautera à l'instruction suivante, car nous avons mis l'offset à 0.

Dans cette instruction, l'offset est ajouté à l'adresse de l'instruction suivante. Donc si l'offset est 0, le saut va transférer l'exécution à l'instruction suivante.

- Le troisième saut est remplacé par JMP comme nous l'avons fait pour le premier, il sera donc toujours effectué.

Voici le code modifié:

```
Hiew: 7_1.exe
   C:\Polygon\ollydbg\7_1.exe         FWO EDITMODE          a32 PE  00000434 Hiew 8.02 (c)SEN
00000400: 55                         push        ebp
00000401: 8BEC                       mov         ebp,esp
00000403: 8B4508                     mov         eax,[ebp][8]
00000406: 3B450C                     cmp         eax,[ebp][00C]
00000409: EB0D                       jmps        000000418
0000040B: 6800B04000                 push        00040B000 ;' @ '
00000410: E8AA000000                 call        0000004BF
00000415: 83C404                     add         esp,4
00000418: 8B4D08                     mov         ecx,[ebp][8]
0000041B: 3B4D0C                     cmp         ecx,[ebp][00C]
0000041E: 7500                       jnz         000000420
00000420: 6808B04000                 push        00040B008 ;' @ '
00000425: E895000000                 call        0000004BF
0000042A: 83C404                     add         esp,4
0000042D: 8B5508                     mov         edx,[ebp][8]
00000430: 3B550C                     cmp         edx,[ebp][00C]
00000433: EB0D                       jmps        000000442
00000435: 6810B04000                 push        00040B010 ;' @ '
0000043A: E880000000                 call        0000004BF
0000043F: 83C404                     add         esp,4
00000442: 5D                         pop         ebp
00000443: C3                         retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
00000444: CC                         int         3
00000445: CC                         int         3
00000446: CC                         int         3
00000447: CC                         int         3
00000448: CC                         int         3
1      2NOPs  3      4      5      6      7      8Table  9      10     11     12
```

Fig. 1.41: Hiew: modifions la fonction `f_unsigned()`

Si nous oublions de modifier une de ces sauts conditionnels, plusieurs appels à `printf()` pourraient être faits, alors que nous voulons qu'un seul soit exécuté.

GCC sans optimisation

GCC 4.4.1 sans optimisation produit presque le même code, mais avec `puts()` (1.5.3 on page 29) à la place de `printf()`.

GCC avec optimisation

Le lecteur attentif pourrait demander pourquoi exécuter CMP plusieurs fois, si les flags ont les mêmes valeurs après l'exécution ?

Peut-être que l'optimiseur de de MSVC ne peut pas faire cela, mais celui de GCC 4.8.1 peut aller plus loin:

Listing 1.112 : GCC 4.8.1 f_signed()

```
f_signed:
        mov     eax, DWORD PTR [esp+8]
        cmp     DWORD PTR [esp+4], eax
        jg      .L6
        je      .L7
        jge     .L1
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC2 ; "a<b"
        jmp     puts
.L6:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0 ; "a>b"
        jmp     puts
.L1:
        rep ret
.L7:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1 ; "a==b"
        jmp     puts
```

Nous voyons ici `JMP puts` au lieu de `CALL puts / RETN`.

Ce genre de truc sera expliqué plus loin: 1.21.1 on page 205.

Ce genre de code x86 est plutôt rare. Il semble que MSVC 2012 ne puisse pas générer un tel code. D'un autre côté, les programmeurs en langage d'assemblage sont parfaitement conscients du fait que les instructions `Jcc` peuvent être empilées.

Donc si vous voyez ce genre d'empilement, il est très probable que le code a été écrit à la main.

La fonction `f_unsigned()` n'est pas si esthétiquement courte:

Listing 1.113 : GCC 4.8.1 f_unsigned()

```
f_unsigned:
        push    esi
        push    ebx
        sub     esp, 20
        mov     esi, DWORD PTR [esp+32]
        mov     ebx, DWORD PTR [esp+36]
        cmp     esi, ebx
        ja      .L13
        cmp     esi, ebx  ; cette instruction peut être supprimée
        je      .L14
.L10:
        jb      .L15
        add     esp, 20
        pop     ebx
        pop     esi
        ret
.L15:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
.L13:
```

```
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
        call    puts
        cmp     esi, ebx
        jne     .L10
.L14:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
```

Néanmoins, il y a deux instructions CMP au lieu de trois.

Donc les algorithmes d'optimisation de GCC 4.8.1 ne sont probablement pas encore parfaits.

ARM

ARM 32-bit

avec optimisation Keil 6/2013 (Mode ARM)

Listing 1.114 : avec optimisation Keil 6/2013 (Mode ARM)

```
.text:000000B8                          EXPORT f_signed
.text:000000B8                  f_signed                ; CODE XREF: main+C
.text:000000B8 70 40 2D E9              STMFD   SP!, {R4-R6,LR}
.text:000000BC 01 40 A0 E1              MOV     R4, R1
.text:000000C0 04 00 50 E1              CMP     R0, R4
.text:000000C4 00 50 A0 E1              MOV     R5, R0
.text:000000C8 1A 0E 8F C2              ADRGT   R0, aAB         ; "a>b\n"
.text:000000CC A1 18 00 CB              BLGT    __2printf
.text:000000D0 04 00 55 E1              CMP     R5, R4
.text:000000D4 67 0F 8F 02              ADREQ   R0, aAB_0       ; "a==b\n"
.text:000000D8 9E 18 00 0B              BLEQ    __2printf
.text:000000DC 04 00 55 E1              CMP     R5, R4
.text:000000E0 70 80 BD A8              LDMGEFD SP!, {R4-R6,PC}
.text:000000E4 70 40 BD E8              LDMFD   SP!, {R4-R6,LR}
.text:000000E8 19 0E 8F E2              ADR     R0, aAB_1       ; "a<b\n"
.text:000000EC 99 18 00 EA              B       __2printf
.text:000000EC                  ; End of function f_signed
```

Beaucoup d'instructions en mode ARM ne peuvent être exécutées que lorsque certains flags sont mis. E.g, ceci est souvent utilisé lorsque l'on compare les nombres

Par exemple, l'instruction ADD est en fait appelée ADDAL en interne, où AL signifie *Always*, i.e., toujours exécuter. Les prédicats sont encodés dans les 4 bits du haut des instructions ARM 32-bit. (*condition field*). L'instruction de saut inconditionnel B est en fait conditionnelle et encodée comme toutes les autres instructions de saut

conditionnel, mais a AL dans le *champ de condition*, et *s'exécute toujours* (ALways), ignorants les flags.

L'instruction ADRGT fonctionne comme ADR mais ne s'exécute que dans le cas où l'instruction CMP précédente a trouvé un des nombres plus grand que l'autre, en comparant les deux (*Greater Than*).

L'instruction BLGT se comporte exactement comme BL et n'est effectuée que si le résultat de la comparaison était *Greater Than* (plus grand). ADRGT écrit un pointeur sur la chaîne a>b\n dans R0 et BLGT appelle printf(). Donc, les instructions suffixées par -GT ne sont exécutées que si la valeur dans R0 (qui est a) est plus grande que la valeur dans R4 (qui est b).

En avançant, nous voyons les instructions ADREQ et BLEQ. Elles se comportent comme ADR et BL, mais ne sont exécutées que si les opérandes étaient égaux lors de la dernière comparaison. Un autre CMP se trouve avant elles (car l'exécution de printf() pourrait avoir modifiée les flags).

Ensuite nous voyons LDMGEFD, cette instruction fonctionne comme LDMFD[92], mais n'est exécutée que si l'une des valeurs est supérieure ou égale à l'autre (*Greater or Equal*).
L'instruction LDMGEFD SP!, {R4-R6,PC} se comporte comme une fonction épilogue, mais elle ne sera exécutée que si $a >= b$, et seulement lorsque l'exécution de la fonction se terminera.

Mais si cette condition n'est pas satisfaite, i.e., $a < b$, alors le flux d'exécution continue à l'instruction suivante, «LDMFD SP!, {R4-R6,LR}», qui est aussi un épilogue de la fonction. Cette instruction ne restaure pas seulement l'état des registres R4-R6, mais aussi LR au lieu de PC, donc il ne retourne pas de la fonction. Les deux dernières instructions appellent printf() avec la chaîne «a<b\n» comme unique argument. Nous avons déjà examiné un saut inconditionnel à la fonction printf() au lieu d'un appel avec retour dans «printf() avec plusieurs arguments» section (1.11.2 on page 75).

f_unsigned est très similaire, à part les instructions ADRHI, BLHI, et LDMCSFD utilisées ici, ces prédicats (*HI = Unsigned higher, CS = Carry Set (greater than or equal)*) sont analogues à ceux examinés avant, mais pour des valeurs non signées.

Il n'y a pas grand chose de nouveau pour nous dans la fonction main() :

Listing 1.115 : main()

```
.text:00000128                  EXPORT main
.text:00000128             main
.text:00000128 10 40 2D E9      STMFD   SP!, {R4,LR}
.text:0000012C 02 10 A0 E3      MOV     R1, #2
.text:00000130 01 00 A0 E3      MOV     R0, #1
.text:00000134 DF FF FF EB      BL      f_signed
.text:00000138 02 10 A0 E3      MOV     R1, #2
.text:0000013C 01 00 A0 E3      MOV     R0, #1
.text:00000140 EA FF FF EB      BL      f_unsigned
.text:00000144 00 00 A0 E3      MOV     R0, #0
.text:00000148 10 80 BD E8      LDMFD   SP!, {R4,PC}
.text:00000148             ; End of function main
```

[92] LDMFD

C’est ainsi que vous pouvez vous débarrasser des sauts conditionnels en mode ARM.

Pourquoi est-ce que c’est si utile? Lire ici: 2.4.1 on page 589.

Il n’y a pas de telle caractéristique en x86, exceptée l’instruction CMOVcc, qui est comme un MOV, mais effectuée seulement lorsque certains flags sont mis, en général mis par CMP.

avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.116 : avec optimisation Keil 6/2013 (Mode Thumb)

```
.text:00000072             f_signed ; CODE XREF: main+6
.text:00000072 70 B5         PUSH    {R4-R6,LR}
.text:00000074 0C 00         MOVS    R4, R1
.text:00000076 05 00         MOVS    R5, R0
.text:00000078 A0 42         CMP     R0, R4
.text:0000007A 02 DD         BLE     loc_82
.text:0000007C A4 A0         ADR     R0, aAB         ; "a>b\n"
.text:0000007E 06 F0 B7 F8   BL      __2printf
.text:00000082
.text:00000082             loc_82 ; CODE XREF: f_signed+8
.text:00000082 A5 42         CMP     R5, R4
.text:00000084 02 D1         BNE     loc_8C
.text:00000086 A4 A0         ADR     R0, aAB_0       ; "a==b\n"
.text:00000088 06 F0 B2 F8   BL      __2printf
.text:0000008C
.text:0000008C             loc_8C ; CODE XREF: f_signed+12
.text:0000008C A5 42         CMP     R5, R4
.text:0000008E 02 DA         BGE     locret_96
.text:00000090 A3 A0         ADR     R0, aAB_1       ; "a<b\n"
.text:00000092 06 F0 AD F8   BL      __2printf
.text:00000096
.text:00000096             locret_96 ; CODE XREF: f_signed+1C
.text:00000096 70 BD         POP     {R4-R6,PC}
.text:00000096             ; End of function f_signed
```

En mode Thumb, seules les instructions B peuvent être complétées par un *condition codes*, (code de condition) donc le code Thumb paraît plus ordinaire.

BLE est un saut conditionnel normal *Less than or Equal* (inférieur ou égal), BNE—*Not Equal* (non égal), BGE—*Greater than or Equal* (supérieur ou égal).

f_unsigned est similaire, seules d’autres instructions sont utilisées pour travailler avec des valeurs non-signées: BLS (*Unsigned lower or same* non signée, inférieur ou égal) et BCS (*Carry Set (Greater than or equal)* supérieur ou égal).

ARM64: GCC (Linaro) 4.9 avec optimisation

Listing 1.117 : f_signed()

```
f_signed:
```

```
; W0=a, W1=b
        cmp     w0, w1
        bgt     .L19    ; Branch if Greater Than
                  ; branchement is supérieur (a>b)
        beq     .L20    ; Branch if Equal
            ; branchement si égal (a==b)
        bge     .L15    ; Branch if Greater than or Equal
            ; branchement si supérieur ou égal (a>=b) (impossible ici)
        ; a<b
        adrp    x0, .LC11       ; "a<b"
        add     x0, x0, :lo12:.LC11
        b       puts
.L19:
        adrp    x0, .LC9        ; "a>b"
        add     x0, x0, :lo12:.LC9
        b       puts
.L15:   ; impossible d'arriver ici
        ret
.L20:
        adrp    x0, .LC10       ; "a==b"
        add     x0, x0, :lo12:.LC10
        b       puts
```

Listing 1.118 : f_unsigned()

```
f_unsigned:
        stp     x29, x30, [sp, -48]!
; W0=a, W1=b
        cmp     w0, w1
        add     x29, sp, 0
        str     x19, [sp,16]
        mov     w19, w0
        bhi     .L25    ; Branch if HIgher
                ; branchement si supérieur (a>b)
        cmp     w19, w1
        beq     .L26    ; Branch if Equal
                  ; branchement si égal (a==b)
.L23:
        bcc     .L27    ; Branch if Carry Clear
                  ; branchement si le flag de retenue est à zéro (si
    inférieur) (a<b)
; épilogue de la fonction, impossible d'arriver ici
        ldr     x19, [sp,16]
        ldp     x29, x30, [sp], 48
        ret
.L27:
        ldr     x19, [sp,16]
        adrp    x0, .LC11       ; "a<b"
        ldp     x29, x30, [sp], 48
        add     x0, x0, :lo12:.LC11
        b       puts
.L25:
        adrp    x0, .LC9        ; "a>b"
        str     x1, [x29,40]
```

```
        add     x0, x0, :lo12:.LC9
        bl      puts
        ldr     x1, [x29,40]
        cmp     w19, w1
        bne     .L23    ; Branch if Not Equal
            ; branchement si non égal
.L26:
        ldr     x19, [sp,16]
        adrp    x0, .LC10       ; "a==b"
        ldp     x29, x30, [sp], 48
        add     x0, x0, :lo12:.LC10
        b       puts
```

Les commentaires ont été ajoutés par l'auteur de ce livre. Ce qui frappe ici, c'est que le compilateur n'est pas au courant que certaines conditions ne sont pas possible du tout, donc il y a du code mort par endroit, qui ne sera jamais exécuté.

Exercice

Essayez d'optimiser manuellement la taille de ces fonctions, en supprimant les instructions redondantes, sans en ajouter de nouvelles.

MIPS

Une des caractéristiques distinctives de MIPS est l'absence de flag. Apparemment, cela a été fait pour simplifier l'analyse des dépendances de données.

Il y a des instructions similaires à SETcc en x86: SLT («Set on Less Than» : mettre si plus petit que, version signée) et SLTU (version non signée). Ces instructions mettent le registre de destination à 1 si la condition est vraie ou à 0 autrement.

Le registre de destination est ensuite testé avec BEQ («Branch on Equal» branchement si égal) ou BNE («Branch on Not Equal» branchement si non égal) et un saut peut survenir. Donc, cette paire d'instructions doit être utilisée en MIPS pour comparer et effectuer un branchement. Essayons avec la version signée de notre fonction:

Listing 1.119 : GCC 4.4.5 sans optimisation (IDA)

```
.text:00000000 f_signed:  # CODE XREF: main+18
.text:00000000
.text:00000000 var_10  = -0x10
.text:00000000 var_8   = -8
.text:00000000 var_4   = -4
.text:00000000 arg_0   =  0
.text:00000000 arg_4   =  4
.text:00000000
.text:00000000         addiu   $sp, -0x20
.text:00000004         sw      $ra, 0x20+var_4($sp)
.text:00000008         sw      $fp, 0x20+var_8($sp)
.text:0000000C         move    $fp, $sp
.text:00000010         la      $gp, __gnu_local_gp
.text:00000018         sw      $gp, 0x20+var_10($sp)
```

```
; stocker les valeurs en entrée sur la pile locale:
.text:0000001C         sw      $a0, 0x20+arg_0($fp)
.text:00000020         sw      $a1, 0x20+arg_4($fp)
; reload them.
.text:00000024         lw      $v1, 0x20+arg_0($fp)
.text:00000028         lw      $v0, 0x20+arg_4($fp)
; $v0=b
; $v1=a
.text:0000002C         or      $at, $zero ; NOP
; ceci est une pseudo-instructions. en fait, c'est "slt $v0,$v0,$v1" ici.
; donc $v0 sera mis à 1 si $v0<$v1 (b<a) ou à 0 autrement:
.text:00000030         slt     $v0, $v1
; saut en loc_5c, si la condition n'est pas vraie.
; ceci est une pseudo-instruction. en fait, c'est "beq $v0,$zero,loc_5c" ici:
.text:00000034         beqz    $v0, loc_5C
; afficher "a>b" et terminer
.text:00000038         or      $at, $zero ; slot de délai de branchement, NOP
.text:0000003C         lui     $v0, (unk_230 >> 16) # "a>b"
.text:00000040         addiu   $a0, $v0, (unk_230 & 0xFFFF) # "a>b"
.text:00000044         lw      $v0, (puts & 0xFFFF)($gp)
.text:00000048         or      $at, $zero ; NOP
.text:0000004C         move    $t9, $v0
.text:00000050         jalr    $t9
.text:00000054         or      $at, $zero ; slot de délai de branchement, NOP
.text:00000058         lw      $gp, 0x20+var_10($fp)
.text:0000005C
.text:0000005C loc_5C:                             # CODE XREF: f_signed+34
.text:0000005C         lw      $v1, 0x20+arg_0($fp)
.text:00000060         lw      $v0, 0x20+arg_4($fp)
.text:00000064         or      $at, $zero ; NOP
; tester si a==b, sauter en loc_90 si ce n'est pas vrai:
.text:00000068         bne     $v1, $v0, loc_90
.text:0000006C         or      $at, $zero ; slot de délai de branchement, NOP
; la condition est vraie, donc afficher "a==b" et terminer:
.text:00000070         lui     $v0, (aAB >> 16)  # "a==b"
.text:00000074         addiu   $a0, $v0, (aAB & 0xFFFF)  # "a==b"
.text:00000078         lw      $v0, (puts & 0xFFFF)($gp)
.text:0000007C         or      $at, $zero ; NOP
.text:00000080         move    $t9, $v0
.text:00000084         jalr    $t9
.text:00000088         or      $at, $zero ; slot de délai de branchement, NOP
.text:0000008C         lw      $gp, 0x20+var_10($fp)
.text:00000090
.text:00000090 loc_90:                             # CODE XREF: f_signed+68
.text:00000090         lw      $v1, 0x20+arg_0($fp)
.text:00000094         lw      $v0, 0x20+arg_4($fp)
.text:00000098         or      $at, $zero ; NOP
; tester si $v1<$v0 (a<b), mettre $v0 à 1 si la condition est vraie:
.text:0000009C         slt     $v0, $v1, $v0
; si la condition n'est pas vraie (i.e., $v0==0), sauter en loc_c8:
.text:000000A0         beqz    $v0, loc_C8
.text:000000A4         or      $at, $zero ; slot de délai de branchement, NOP
; la condition est vraie, afficher "a<b" et terminer
```

```
.text:000000A8         lui     $v0, (aAB_0 >> 16)  # "a<b"
.text:000000AC         addiu   $a0, $v0, (aAB_0 & 0xFFFF)  # "a<b"
.text:000000B0         lw      $v0, (puts & 0xFFFF)($gp)
.text:000000B4         or      $at, $zero ; NOP
.text:000000B8         move    $t9, $v0
.text:000000BC         jalr    $t9
.text:000000C0         or      $at, $zero ; slot de délai de branchement, NOP
.text:000000C4         lw      $gp, 0x20+var_10($fp)
.text:000000C8
; toutes les 3 conditions étaient fausses, donc simplement terminer:
.text:000000C8 loc_C8:                                  # CODE XREF:
    f_signed+A0
.text:000000C8         move    $sp, $fp
.text:000000CC         lw      $ra, 0x20+var_4($sp)
.text:000000D0         lw      $fp, 0x20+var_8($sp)
.text:000000D4         addiu   $sp, 0x20
.text:000000D8         jr      $ra
.text:000000DC         or      $at, $zero ; slot de délai de branchement, NOP
.text:000000DC  # Fin de la fonction f_signed
```

SLT REG0, REG0, REG1 est réduit par IDA à sa forme plus courte:
SLT REG0, REG1.

Nous voyons également ici la pseudo instruction BEQZ («Branch if Equal to Zero» branchement si égal à zéro),
qui est en fait BEQ REG, $ZERO, LABEL.

La version non signée est la même, mais SLTU (version non signée, d'où le «U» de unsigned) est utilisée au lieu de SLT :

Listing 1.120 : GCC 4.4.5 sans optimisation (IDA)

```
.text:000000E0 f_unsigned:  # CODE XREF: main+28
.text:000000E0
.text:000000E0 var_10  = -0x10
.text:000000E0 var_8   = -8
.text:000000E0 var_4   = -4
.text:000000E0 arg_0   =  0
.text:000000E0 arg_4   =  4
.text:000000E0
.text:000000E0         addiu   $sp, -0x20
.text:000000E4         sw      $ra, 0x20+var_4($sp)
.text:000000E8         sw      $fp, 0x20+var_8($sp)
.text:000000EC         move    $fp, $sp
.text:000000F0         la      $gp, __gnu_local_gp
.text:000000F8         sw      $gp, 0x20+var_10($sp)
.text:000000FC         sw      $a0, 0x20+arg_0($fp)
.text:00000100         sw      $a1, 0x20+arg_4($fp)
.text:00000104         lw      $v1, 0x20+arg_0($fp)
.text:00000108         lw      $v0, 0x20+arg_4($fp)
.text:0000010C         or      $at, $zero
.text:00000110         sltu    $v0, $v1
.text:00000114         beqz    $v0, loc_13C
.text:00000118         or      $at, $zero
```

```
.text:0000011C         lui     $v0, (unk_230 >> 16)
.text:00000120         addiu   $a0, $v0, (unk_230 & 0xFFFF)
.text:00000124         lw      $v0, (puts & 0xFFFF)($gp)
.text:00000128         or      $at, $zero
.text:0000012C         move    $t9, $v0
.text:00000130         jalr    $t9
.text:00000134         or      $at, $zero
.text:00000138         lw      $gp, 0x20+var_10($fp)
.text:0000013C
.text:0000013C loc_13C:                        # CODE XREF: f_unsigned+34
.text:0000013C         lw      $v1, 0x20+arg_0($fp)
.text:00000140         lw      $v0, 0x20+arg_4($fp)
.text:00000144         or      $at, $zero
.text:00000148         bne     $v1, $v0, loc_170
.text:0000014C         or      $at, $zero
.text:00000150         lui     $v0, (aAB >> 16)  # "a==b"
.text:00000154         addiu   $a0, $v0, (aAB & 0xFFFF)  # "a==b"
.text:00000158         lw      $v0, (puts & 0xFFFF)($gp)
.text:0000015C         or      $at, $zero
.text:00000160         move    $t9, $v0
.text:00000164         jalr    $t9
.text:00000168         or      $at, $zero
.text:0000016C         lw      $gp, 0x20+var_10($fp)
.text:00000170
.text:00000170 loc_170:                        # CODE XREF: f_unsigned+68
.text:00000170         lw      $v1, 0x20+arg_0($fp)
.text:00000174         lw      $v0, 0x20+arg_4($fp)
.text:00000178         or      $at, $zero
.text:0000017C         sltu    $v0, $v1, $v0
.text:00000180         beqz    $v0, loc_1A8
.text:00000184         or      $at, $zero
.text:00000188         lui     $v0, (aAB_0 >> 16)  # "a<b"
.text:0000018C         addiu   $a0, $v0, (aAB_0 & 0xFFFF)  # "a<b"
.text:00000190         lw      $v0, (puts & 0xFFFF)($gp)
.text:00000194         or      $at, $zero
.text:00000198         move    $t9, $v0
.text:0000019C         jalr    $t9
.text:000001A0         or      $at, $zero
.text:000001A4         lw      $gp, 0x20+var_10($fp)
.text:000001A8
.text:000001A8 loc_1A8:                        # CODE XREF: f_unsigned+A0
.text:000001A8         move    $sp, $fp
.text:000001AC         lw      $ra, 0x20+var_4($sp)
.text:000001B0         lw      $fp, 0x20+var_8($sp)
.text:000001B4         addiu   $sp, 0x20
.text:000001B8         jr      $ra
.text:000001BC         or      $at, $zero
.text:000001BC  # End of function f_unsigned
```

1.18.2 Calcul de valeur absolue

Une fonction simple:

```
int my_abs (int i)
{
        if (i<0)
                return -i;
        else
                return i;
};
```

MSVC avec optimisation

Ceci est le code généré habituellement:

Listing 1.121 : MSVC 2012 x64 avec optimisation

```
i$ = 8
my_abs  PROC
; ECX = valeur en entrée
        test    ecx, ecx
; tester le signe de la valeur en entrée
; sauter l'instruction NEG si le signe est positif
        jns     SHORT $LN2@my_abs
; inverser la valeur
        neg     ecx
$LN2@my_abs:
; copier le résultat dans EAX:
        mov     eax, ecx
        ret     0
my_abs  ENDP
```

GCC 4.9 génère en gros le même code:

avec optimisation Keil 6/2013 : Mode Thumb

Listing 1.122 : avec optimisation Keil 6/2013 : Mode Thumb

```
my_abs PROC
        CMP      r0,#0
; est-ce que la valeur en entrée est égale ou plus grande que zéro?
; si oui, sauter l'instruction RSBS
        BGE      |L0.6|
; soustraire la valeur en entrée de 0:
        RSBS     r0,r0,#0
|L0.6|
        BX       lr
        ENDP
```

Il manque une instruction de négation en ARM, donc le compilateur Keil utilise l'instruction «Reverse Subtract», qui soustrait la valeur du registre de l'opérande.

avec optimisation Keil 6/2013 : Mode ARM

Il est possible d'ajouter un code de condition à certaines instructions en mode ARM, c'est donc ce que fait le compilateur Keil:

Listing 1.123 : avec optimisation Keil 6/2013 : Mode ARM

```
my_abs PROC
        CMP      r0,#0
; exécuter l'instruction "Reverse Subtract" seulement si la valeur en entrée
; est plus petite que 0:
        RSBLT    r0,r0,#0
        BX       lr
        ENDP
```

Maintenant, il n'y a plus de saut conditionnel et c'est mieux: 2.4.1 on page 589.

GCC 4.9 sans optimisation (ARM64)

ARM64 possède l'instruction NEG pour effectuer la négation:

Listing 1.124 : GCC 4.9 avec optimisation (ARM64)

```
my_abs:
        sub     sp, sp, #16
        str     w0, [sp,12]
        ldr     w0, [sp,12]
; comparer la valeur en entrée avec le contenu du registre WZR
; (qui contient toujours zéro)
        cmp     w0, wzr
        bge     .L2
        ldr     w0, [sp,12]
        neg     w0, w0
        b       .L3
.L2:
        ldr     w0, [sp,12]
.L3:
        add     sp, sp, 16
        ret
```

MIPS

Listing 1.125 : GCC 4.4.5 avec optimisation (IDA)

```
my_abs:
; saut si $a0<0:
                bltz    $a0, locret_10
; simplement renvoyer la valeur en entrée ($a0) dans $v0:
                move    $v0, $a0
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
locret_10:
; prendre l'opposée de la valeur entrée et la stocker dans $v0:
```

```
                jr      $ra
; ceci est une pseudo-instruction. En fait, ceci est "subu $v0,$zero,$a0"
    ($v0=0-$a0)
                negu    $v0, $a0
```

Nous voyons ici une nouvelle instruction: BLTZ («Branch if Less Than Zero » branchement si plus petit que zéro).

Il y a aussi la pseudo-instruction NEGU, qui effectue une soustraction à zéro. Le suffixe «U » dans les deux instructions SUBU et NEGU indique qu'aucune exception ne sera levée en cas de débordement de la taille d'un entier.

Version sans branchement?

Vous pouvez aussi avoir une version sans branchement de ce code. Ceci sera revu plus tard: 3.16 on page 664.

1.18.3 Opérateur conditionnel ternaire

L'opérateur conditionnel ternaire en C/C++ a la syntaxe suivante:

```
expression ? expression : expression
```

Voici un exemple:

```
const char* f (int a)
{
        return a==10 ? "it is ten" : "it is not ten";
};
```

x86

Les vieux compilateurs et ceux sans optimisation génèrent du code assembleur comme si des instructions if/else avaient été utilisées:

Listing 1.126 : MSVC 2008 sans optimisation

```
$SG746  DB      'it is ten', 00H
$SG747  DB      'it is not ten', 00H

tv65 = -4 ; ceci sera utilisé comme variable temporaire
_a$ = 8
_f      PROC
        push    ebp
        mov     ebp, esp
        push    ecx
; comparer la valeur en entrée avec 10
        cmp     DWORD PTR _a$[ebp], 10
; sauter en $LN3@f si non égal
        jne     SHORT $LN3@f
; stocker le pointeur sur la chaîne dans la variable temporaire:
        mov     DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten'
```

```
; sauter à la sortie
        jmp     SHORT $LN4@f
$LN3@f:
; stocker le pointeur sur la chaîne dans la variable temporaire:
        mov     DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not ten'
$LN4@f:
; ceci est la sortie.
; copier le pointeur sur la chaîne depuis la variable temporaire dans EAX.
        mov     eax, DWORD PTR tv65[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

Listing 1.127 : MSVC 2008 avec optimisation

```
$SG792  DB      'it is ten', 00H
$SG793  DB      'it is not ten', 00H

_a$ = 8 ; taille = 4
_f      PROC
; comparer la valeur en entrée avec 10
        cmp     DWORD PTR _a$[esp-4], 10
        mov     eax, OFFSET $SG792 ; 'it is ten'
; sauter en $LN4@f si égal
        je      SHORT $LN4@f
        mov     eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
        ret     0
_f      ENDP
```

Les nouveaux compilateurs sont plus concis:

Listing 1.128 : MSVC 2012 x64 avec optimisation

```
$SG1355 DB      'it is ten', 00H
$SG1356 DB      'it is not ten', 00H

a$ = 8
f       PROC
; charger les pointeurs sur les deux chaînes
        lea     rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
        lea     rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
; comparer la valeur en entrée avec 10
        cmp     ecx, 10
; si égal, copier la valeur dans RDX ("it is ten")
; si non, ne rien faire. le pointeur sur la chaîne
; "it is not ten" est encore dans RAX à ce stade.
        cmove   rax, rdx
        ret     0
f       ENDP
```

GCC 4.8 avec optimisation pour x86 utilise également l'instruction CMOVcc, tandis que GCC 4.8 sans optimisation utilise des sauts conditionnels.

ARM

Keil avec optimisation pour le mode ARM utilise les instructions conditionnelles ADRcc :

Listing 1.129 : avec optimisation Keil 6/2013 (Mode ARM)

```
f PROC
; comparer la valeur en entrée avec 10
        CMP      r0,#0xa
; si le résultat de la comparaison est égal (EQual), copier le pointeur sur
    la chaîne
; "it is ten" dans R0
        ADREQ    r0,|L0.16| ; "it is ten"
; si le résultat de la comparaison est non égal (Not EQual), copier le
    pointeur sur la chaîne
; "it is not ten" dans R0
        ADRNE    r0,|L0.28| ; "it is not ten"
        BX       lr
        ENDP

|L0.16|
        DCB      "it is ten",0
|L0.28|
        DCB      "it is not ten",0
```

Sans intervention manuelle, les deux instructions ADREQ et ADRNE ne peuvent être exécutées lors de la même exécution.

Keil avec optimisation pour le mode Thumb à besoin d'utiliser des instructions de saut conditionnel, puisqu'il n'y a pas d'instruction qui supporte le flag conditionnel.

Listing 1.130 : avec optimisation Keil 6/2013 (Mode Thumb)

```
f PROC
; comparer la valeur entrée avec 10
        CMP      r0,#0xa
; sauter en |L0.8| si égal (EQual)
        BEQ      |L0.8|
        ADR      r0,|L0.12| ; "it is not ten"
        BX       lr
|L0.8|
        ADR      r0,|L0.28| ; "it is ten"
        BX       lr
        ENDP

|L0.12|
        DCB      "it is not ten",0
|L0.28|
        DCB      "it is ten",0
```

ARM64

GCC (Linaro) 4.9 avec optimisation pour ARM64 utilise aussi des sauts conditionnels:

Listing 1.131 : GCC (Linaro) 4.9 avec optimisation

```
f:
        cmp     x0, 10
        beq     .L3             ; branchement si égal
        adrp    x0, .LC1        ; "it is ten"
        add     x0, x0, :lo12:.LC1
        ret
.L3:
        adrp    x0, .LC0        ; "it is not ten"
        add     x0, x0, :lo12:.LC0
        ret
.LC0:
        .string "it is ten"
.LC1:
        .string "it is not ten"
```

C'est parce qu'ARM64 n'a pas d'instruction de chargement simple avec le flag conditionnel comme ADRcc en ARM 32-bit ou CMOVcc en x86.

Il a toutefois l'instruction «Conditional SELect» (CSEL)[*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)p390, C5.5], mais GCC 4.9 ne semble pas assez malin pour l'utiliser dans un tel morceau de code.

MIPS

Malheureusement, GCC 4.4.5 pour MIPS n'est pas très malin non plus:

Listing 1.132 : GCC 4.4.5 avec optimisation (résultat en sortie de l'assembleur)

```
$LC0:
        .ascii  "it is not ten\000"
$LC1:
        .ascii  "it is ten\000"
f:
        li      $2,10                   # 0xa
; comparer $a0 et 10, sauter si égal:
        beq     $4,$2,$L2
        nop ; slot de délai de branchement

; charger l'adresse de la chaîne "it is not ten" dans $v0 et sortir:
        lui     $2,%hi($LC0)
        j       $31
        addiu   $2,$2,%lo($LC0)

$L2:
; charger l'adresse de la chaîne "it is ten" dans $v0 et sortir:
        lui     $2,%hi($LC1)
        j       $31
        addiu   $2,$2,%lo($LC1)
```

Récrivons-le à l'aide d'un if/else

```
const char* f (int a)
{
        if (a==10)
                return "it is ten";
        else
                return "it is not ten";
};
```

Curieusement, GCC 4.8 avec l'optimisation a pû utiliser CMOVcc dans ce cas:

Listing 1.133 : GCC 4.8 avec optimisation

```
.LC0:
        .string "it is ten"
.LC1:
        .string "it is not ten"
f:
.LFB0:
; comparer la valeur en entrée avec 10
        cmp     DWORD PTR [esp+4], 10
        mov     edx, OFFSET FLAT:.LC1 ; "it is not ten"
        mov     eax, OFFSET FLAT:.LC0 ; "it is ten"
; si le résultat de la comparaison est Not Equal, copier la valeur de EDX
    dans EAX
; sinon, ne rien faire
        cmovne  eax, edx
        ret
```

Keil avec optimisation génère un code identique à listado.1.129.

Mais MSVC 2012 avec optimisation n'est pas (encore) si bon.

Conclusion

Pourquoi est-ce que les compilateurs qui optimisent essayent de se débarrasser des sauts conditionnels? Voir à ce propos: 2.4.1 on page 589.

1.18.4 Trouver les valeurs minimale et maximale

32-bit

```
int my_max(int a, int b)
{
        if (a>b)
                return a;
        else
                return b;
};

int my_min(int a, int b)
{
        if (a<b)
```

```
                return a;
        else
                return b;
};
```

Listing 1.134 : MSVC 2013 sans optimisation

```
_a$ = 8
_b$ = 12
_my_min PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; comparer A et B:
        cmp     eax, DWORD PTR _b$[ebp]
; sauter si A est supérieur ou égal à B:
        jge     SHORT $LN2@my_min
; recharger A dans EAX si autrement et sauter à la sortie
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_min
        jmp     SHORT $LN3@my_min ; ce JMP est redondant
$LN2@my_min:
; renvoyer B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_min:
        pop     ebp
        ret     0
_my_min ENDP

_a$ = 8
_b$ = 12
_my_max PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; comparer A et B:
        cmp     eax, DWORD PTR _b$[ebp]
; sauter si A est inférieur ou égal à B:
        jle     SHORT $LN2@my_max
; recharger A dans EAX si autrement et sauter à la sortie
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_max
        jmp     SHORT $LN3@my_max ; ce JMP est redondant
$LN2@my_max:
; renvoyer B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_max:
        pop     ebp
        ret     0
_my_max ENDP
```

Ces deux fonctions ne diffèrent que de l'instruction de saut conditionnel: JGE («Jump if Greater or Equal» saut si supérieur ou égal) est utilisée dans la première et JLE

(«Jump if Less or Equal » saut si inférieur ou égal) dans la seconde.

Il y a une instruction JMP en trop dans chaque fonction, que MSVC a probablement mise par erreur.

Sans branchement

Le mode Thumb d'ARM nous rappelle le code x86:

Listing 1.135 : avec optimisation Keil 6/2013 (Mode Thumb)

```
my_max PROC
; R0=A
; R1=B
; comparer A et B:
        CMP      r0,r1
; branchement si A est supérieur à B:
        BGT      |L0.6|
; autrement (A<=B) renvoyer R1 (B):
        MOVS     r0,r1
|L0.6|
; retourner
        BX       lr
        ENDP

my_min PROC
; R0=A
; R1=B
; comparer A et B:
        CMP      r0,r1
; branchement si A est inférieur à B:
        BLT      |L0.14|
; autrement (A>=B) renvoyer R1 (B):
        MOVS     r0,r1
|L0.14|
; retourner
        BX       lr
        ENDP
```

Les fonctions diffèrent au niveau de l'instruction de branchement: BGT et BLT. Il est possible d'utiliser le suffixe conditionnel en mode ARM, donc le code est plus court.

MOVcc n'est exécutée que si la condition est remplie:

Listing 1.136 : avec optimisation Keil 6/2013 (Mode ARM)

```
my_max PROC
; R0=A
; R1=B
; comparer A et B:
        CMP      r0,r1
; renvoyer B au lieu de A en copiant B dans R0
; cette instruction ne s'exécutera que si A<=B (en effet, LE Less or Equal,
    inférieur ou égal)
```

```
; si l'instruction n'est pas exécutée (dans le cas où A>B),
; A est toujours dans le registre R0
        MOVLE    r0,r1
        BX       lr
        ENDP

my_min PROC
; R0=A
; R1=B
; comparer A et B:
        CMP      r0,r1
; renvoyer B au lieu de A en copiant B dans R0
; cette instruction ne s'exécutera que si A>=B (GE Greater or Equal,
    supérieur ou égal)
; si l'instruction n'est pas exécutée (dans le cas où A<B),
; A est toujours dans le registre R0
        MOVGE    r0,r1
        BX       lr
        ENDP
```

GCC 4.8.1 avec optimisation et MSVC 2013 avec optimisation peuvent utiliser l'instruction CMOVcc, qui est analogue à MOVcc en ARM:

Listing 1.137 : MSVC 2013 avec optimisation

```
my_max:
        mov     edx, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; comparer A et B:
        cmp     edx, eax
; si A>=B, charger la valeur A dans EAX
; l'instruction ne fait rien autrement (si A<B)
        cmovge  eax, edx
        ret

my_min:
        mov     edx, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; comparer A et B:
        cmp     edx, eax
; si A<=B, charger la valeur A dans EAX
; l'instruction ne fait rien autrement (si A>B)
        cmovle  eax, edx
        ret
```

64-bit

```
#include <stdint.h>
```

```
int64_t my_max(int64_t a, int64_t b)
{
        if (a>b)
                return a;
        else
                return b;
};

int64_t my_min(int64_t a, int64_t b)
{
        if (a<b)
                return a;
        else
                return b;
};
```

Il y a beaucoup de code inutile qui embrouille, mais il est compréhensible:

Listing 1.138 : GCC 4.9.1 ARM64 sans optimisation

```
my_max:
        sub     sp, sp, #16
        str     x0, [sp,8]
        str     x1, [sp]
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        cmp     x1, x0
        ble     .L2
        ldr     x0, [sp,8]
        b       .L3
.L2:
        ldr     x0, [sp]
.L3:
        add     sp, sp, 16
        ret

my_min:
        sub     sp, sp, #16
        str     x0, [sp,8]
        str     x1, [sp]
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        cmp     x1, x0
        bge     .L5
        ldr     x0, [sp,8]
        b       .L6
.L5:
        ldr     x0, [sp]
.L6:
        add     sp, sp, 16
        ret
```

Sans branchement

Il n'y a pas besoin de lire les arguments dans la pile, puisqu'ils sont déjà dans les registres:

Listing 1.139 : GCC 4.9.1 x64 avec optimisation

```
my_max:
; RDI=A
; RSI=B
; comparer A et B:
        cmp     rdi, rsi
; préparer B pour le renvoyer dans RAX:
        mov     rax, rsi
; si A>=B, mettre A (RDI) dans RAX pour le renvoyer.
; cette instruction ne fait rien autrement (si A<B)
        cmovge  rax, rdi
        ret

my_min:
; RDI=A
; RSI=B
; comparer A et B:
        cmp     rdi, rsi
; préparer B pour le renvoyer dans RAX:
        mov     rax, rsi
; si A<=B, mettre A (RDI) dans RAX pour le renvoyer.
; cette instruction ne fait rien autrement (si A>B)
        cmovle  rax, rdi
        ret
```

MSVC 2013 fait presque la même chose.

ARM64 possède l'instruction CSEL, qui fonctionne comme MOVcc en ARM ou CMOVcc en x86, seul le nom diffère: «Conditional SELect».

Listing 1.140 : GCC 4.9.1 ARM64 avec optimisation

```
my_max:
; X0=A
; X1=B
; comparer A et B:
        cmp     x0, x1
; copier X0 (A) dans X0 si X0>=X1 ou A>=B (Greater or Equal, supérieur ou
    égal)
; copier X1 (B) dans X0 si A<B
        csel    x0, x0, x1, ge
        ret

my_min:
; X0=A
; X1=B
; comparer A et B:
        cmp     x0, x1
```

```
; copier X0 (A) dans X0 si X0<=X1 ou A<=B (Less or Equal, inférieur ou égal)
; copier X1 (B) dans X0 si A>B
        csel    x0, x0, x1, le
        ret
```

MIPS

Malheureusement, GCC 4.4.5 pour MIPS n'est pas si performant:

Listing 1.141 : GCC 4.4.5 avec optimisation (IDA)

```
my_max:
; mettre $v1 à 1 si $a1<$a0, ou l'effacer autrement (si $a1>$a0):
                slt     $v1, $a1, $a0
; sauter, si $v1 est 0 (ou $a1>$a0):
                beqz    $v1, locret_10
; ceci est le slot de délai de branchement
; préparer $a1 dans $v0 si le branchement est pris:
                move    $v0, $a1
; le branchment n'est pas pris, préparer $a0 dans $v0:
                move    $v0, $a0

locret_10:
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP

; la fonction min() est la même, mais les opérandes
; dans l'instruction SLT sont échangés:
my_min:
                slt     $v1, $a0, $a1
                beqz    $v1, locret_28
                move    $v0, $a1
                move    $v0, $a0

locret_28:
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

N'oubliez pas le slot de délai de branchement (*branch delay slots*) : le premier MOVE est exécuté *avant* BEQZ, le second MOVE n'est exécuté que si la branche n'a pas été prise.

1.18.5 Conclusion

x86

Voici le squelette générique d'un saut conditionnel:

Listing 1.142 : x86

```
CMP registre, registre/valeur
Jcc true ; cc=condition code, code de condition
false:
```

```
;... le code qui sera exécuté si le résultat de la comparaison est faux
    (false) ...
JMP exit
true:
;... le code qui sera exécuté si le résultat de la comparaison est vrai
    (true) ...
exit:
```

ARM

Listing 1.143 : ARM

```
CMP registre, registre/valeur
Bcc true ; cc=condition code
false:
;... le code qui sera exécuté si le résultat de la comparaison est faux
    (false) ...
JMP exit
true:
;... le code qui sera exécuté si le résultat de la comparaison est vrai
    (true) ...
exit:
```

MIPS

Listing 1.144 : Teste si égal à zéro (Branch if EQual Zero)

```
BEQZ REG, label
...
```

Listing 1.145 : Teste si plus petit que zéro (Branch if Less Than Zero) en utilisant une pseudo instruction

```
BLTZ REG, label
...
```

Listing 1.146 : Teste si les valeurs sont égales (Branch if EQual)

```
BEQ REG1, REG2, label
...
```

Listing 1.147 : Teste si les valeurs ne sont pas égales (Branch if Not Equal)

```
BNE REG1, REG2, label
...
```

Listing 1.148 : Teste si REG2 est plus petit que REG3 (signé)

```
SLT REG1, REG2, REG3
BEQ REG1, label
...
```

Listing 1.149 : Teste si REG2 est plus petit que REG3 (non signé)

```
SLTU REG1, REG2, REG3
BEQ REG1, label
...
```

Sans branchement

Si le corps d'instruction conditionnelle est très petit, l'instruction de déplacement conditionnel peut être utilisée: MOVcc en ARM (en mode ARM), CSEL en ARM64, CMOVcc en x86.

ARM

Il est possible d'utiliser les suffixes conditionnels pour certaines instructions ARM:

Listing 1.150 : ARM (Mode ARM)

```
CMP registre, registre/valeur
instr1_cc ; cette instruction sera exécutée si le code conditionnel est vrai
    (true)
instr2_cc ; cette autre instruction sera exécutée si cet autre code
    conditionnel est vrai (true)
;... etc.
```

Bien sûr, il n'y a pas de limite au nombre d'instructions avec un suffixe de code conditionnel, tant que les flags du CPU ne sont pas modifiés par l'une d'entre elles.

Le mode Thumb possède l'instruction IT, permettant d'ajouter le suffixe conditionnel pour les quatre instructions suivantes. Lire à ce propos: 1.25.7 on page 334.

Listing 1.151 : ARM (Mode Thumb)

```
CMP registre, registre/valeur
ITEEE EQ ; met ces suffixes: if-then-else-else-else
instr1   ; instruction exécutée si la condition est vraie
instr2   ; instruction exécutée si la condition est fausse
instr3   ; instruction exécutée si la condition est fausse
instr4   ; instruction exécutée si la condition est fausse
```

1.18.6 Exercice

(ARM64) Essayez de récrire le code pour listado.1.131 en supprimant toutes les instructions de saut conditionnel et en utilisant l'instruction CSEL.

1.19 Déplombage de logiciel

La grande majorité des logiciels peuvent être déplombés comme ça — en cherchant l'endroit où la protection est vérifiée, un dongle (8.8 on page 1081), une clef de licence, un numéro de série, etc.

Souvent, ça ressemble à ça:

```
...
call check_protection
jz   all_OK
call message_box_protection_missing
call exit
all_OK:
; proceed
...
```

Donc, si vous voyez un patch (ou “crack”), qui déplombe un logiciel, et que ce patch remplace un ou des octets 0x74/0x75 (JZ/JNZ) par 0xEB (JMP), c’est ça.

Le processus de déplombage de logiciel revient à une recherche de ce JMP.

Il y a aussi les cas où le logiciel vérifie la protection de temps à autre, ceci peut être un dongle, ou un serveur de licence qui peut être interrogé depuis Internet. Dans ce cas, vous devez chercher une fonction qui vérifie la protection. Puis, la modifier, pour y mettre `xor eax, eax / retn`, ou `mov eax, 1 / retn`.

Il est important de comprendre qu’après avoir patché le début d’une fonction, souvent, il y a des octets résiduels qui suivent ces deux instructions. Ces restes consistent en une partie d’une instruction et les instructions suivantes.

Ceci est un cas réel. Le début de la fonction que nous voulons *remplacer* par `return 1;`

Listing 1.152 : Before

```
8BFF                          mov         edi,edi
55                            push        ebp
8BEC                          mov         ebp,esp
81EC68080000                  sub         esp,000000868
A110C00001                    mov         eax,[00100C010]
33C5                          xor         eax,ebp
8945FC                        mov         [ebp][-4],eax
53                            push        ebx
8B5D08                        mov         ebx,[ebp][8]
...
```

Listing 1.153 : After

```
B801000000                    mov         eax,1
C3                            retn
EC                            in          al,dx
68080000A1                    push        0A1000008
10C0                          adc         al,al
0001                          add         [ecx],al
33C5                          xor         eax,ebp
8945FC                        mov         [ebp][-4],eax
53                            push        ebx
8B5D08                        mov         ebx,[ebp][8]
...
```

Quelques instructions incorrectes apparaissent — IN, PUSH, ADC, ADD, après lesquelles, le désassembleur Hiew (que j'ai utilisé) s'est synchronisé et a continué de désassembler le reste.

Ceci n'est pas important — toutes ces instructions qui suivent RETN ne seront jamais exécutées, à moins qu'un saut direct se produise quelque part, et ça ne sera pas possible en général.

Il peut aussi y avoir une variable globale booléenne, un flag indiquant si le logiciel est enregistré ou non.

```
init_etc proc
...
call check_protection_or_license_file
mov  is_demo, eax
...
retn
init_etc endp

...

save_file proc
...
mov  eax, is_demo
cmp  eax, 1
jz   all_OK1

call message_box_it_is_a_demo_no_saving_allowed
retn

:all_OK1
; continuer en sauvant le fichier

...

save_proc endp

somewhere_else proc

mov  eax, is_demo
cmp  eax, 1
jz   all_OK

; contrôler si le programme fonctionne depuis 15 minutes
; sortir si c'est le cas
; ou montrer un écran fixe

:all_OK2
; continuer

somewhere_else endp
```

Le début de la fonction `check_protection_or_license_file()` peut être patché, afin qu'elle renvoie toujours 1, ou, si c'est mieux pour une raison quelconques, toutes les instructions JZ/JNZ peuvent être patchées de même

Plus sur le patching: 11.2.

1.20 Blague de l'arrêt impossible (Windows 7)

Je ne me rappelle pas vraiment comment j'ai découvert la fonction `ExitWindowsEx()` dans le fichier user32.dll de Windows 98 (c'était à la fin des années 1990) J'ai probablement juste remarqué le nom évocateur. Et j'ai ensuite essayé de la *bloquer* en modifiant son début par l'octet 0xC3 byte (RETN).

Le résultat était rigolo: Windows 98 ne pouvait plus être arrêté. Il fallait appuyer sur le bouton reset.

Ces jours, j'ai essayé de faire de même dans Windows 7, qui a été créé presque 10 ans après et qui est basé sur une base Windows NT complètement différente. Quand même, la fonction `ExitWindowsEx()` est présente dans le fichier user32.dll et sert à la même chose.

Premièrement, j'ai arrêté *Windows File Protection* en ajoutant ceci dans la base de registres (sinon Windows restaurerai silencieusement les fichiers système modifiés) :

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon]
"SFCDisable"=dword:ffffff9d
```

Puis, j'ai renommé `c:\windows\system32\user32.dll` en `user32.dll.bak`. J'ai trouvé l' entrée de l'export `ExitWindowsEx()` en utilisant Hiew (IDA peut aider de même) et y ai mis l'octet 0xC3. J'ai redémarré Windows 7 et maintenant on ne peut plus l'arrêter. Les boutons "Restart" et "Logoff" ne fonctionnent plus.

Je ne sais pas si c'est rigolo aujourd'hui ou pas, mais dans le passé, à la fin des années 1990, mon ami a pris le fichier user32.dll patché sur une disquette et l'a copié sur tous les ordinateurs (à portée de main, qui fonctionnaient sous Windows 98 (presque tous)) de son université. Plus aucun ordinateur sous Windows ne pouvait être arrêté après et son professeur d'informatique était rouge. (Espérons qu'il puisse nous pardonner aujourd'hui s'il lit ceci maintenant.)

Si vous faîtes ça, sauvegardez tout. La meilleure idée est de lancer Windows dans une machine virtuelle.

1.21 switch()/case/default

1.21.1 Petit nombre de cas

```
#include <stdio.h>

void f (int a)
```

```
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};
```

x86

MSVC sans optimisation

Résultat (MSVC 2010) :

Listing 1.154 : MSVC 2010

```
tv64 = -4 ; size = 4
_a$ = 8   ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 0
    je     SHORT $LN4@f
    cmp    DWORD PTR tv64[ebp], 1
    je     SHORT $LN3@f
    cmp    DWORD PTR tv64[ebp], 2
    je     SHORT $LN2@f
    jmp    SHORT $LN1@f
$LN4@f:
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN3@f:
    push   OFFSET $SG741 ; 'one', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN2@f:
    push   OFFSET $SG743 ; 'two', 0aH, 00H
    call   _printf
    add    esp, 4
```

```
    jmp    SHORT $LN7@f
$LN1@f:
    push   OFFSET $SG745 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN7@f:
    mov    esp, ebp
    pop    ebp
    ret    0
_f    ENDP
```

Notre fonction avec quelques cas dans switch() est en fait analogue à cette construction:

```
void f (int a)
{
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
};
```

Si nous utilisons switch() avec quelques cas, il est impossible de savoir si il y avait un vrai switch() dans le code source, ou un ensemble de directives if().

Ceci indique que switch() est comme un sucre syntaxique pour un grand nombre de if() imbriqués.

Il n'y a rien de particulièrement nouveau pour nous dans le code généré, à l'exception que le compilateur déplace la variable d'entrée a dans une variable locale temporaire tv64 [93].

Si nous compilons ceci avec GCC 4.4.1, nous obtenons presque le même résultat, même avec le niveau d'optimisation le plus élevé (-O3 option).

MSVC avec optimisation

Maintenant compilons dans MSVC avec l'optimisation (/Ox) : `cl 1.c /Fa1.asm /Ox`

Listing 1.155 : MSVC

```
_a$ = 8 ; size = 4
_f    PROC
    mov    eax, DWORD PTR _a$[esp-4]
    sub    eax, 0
    je     SHORT $LN4@f
```

[93] Les variables locales sur la pile sont préfixées avec tv—c'est ainsi que MSVC appelle les variables internes dont il a besoin.

```
    sub     eax, 1
    je      SHORT $LN3@f
    sub     eax, 1
    je      SHORT $LN2@f
    mov     DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something unknown', 0aH,
    00H
    jmp     _printf
$LN2@f:
    mov     DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H
    jmp     _printf
$LN3@f:
    mov     DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00H
    jmp     _printf
$LN4@f:
    mov     DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, 00H
    jmp     _printf
_f    ENDP
```

Ici, nous voyons quelques hacks moches.

Premièrement: la valeurs de a est mise dans EAX et 0 en est soustrait. Ça semble absurde, mais cela est fait pour vérifier si la valeur dans EAX est 0. Si oui, le flag ZF est mis (e.g. soustraire de 0 est 0) et le premier saut conditionnel JE (*Jump if Equal* saut si égal ou synonyme JZ —*Jump if Zero* saut si zéro) va être effectué et le déroulement du programme passera au label `$LN4@f`, où le message `'zero'` est affiché. Si le premier saut n'est pas effectué, 1 est soustrait de la valeur d'entrée et si à une étape le résultat est 0, le saut correspondant sera effectué.

Et si aucun saut n'est exécuté, l'exécution passera au `printf()` avec comme argument la chaîne
`'something unknown'`.

Deuxièmement: nous voyons quelque chose d'inhabituel pour nous: un pointeur sur une chaîne est mis dans la variable a et ensuite `printf()` est appelé non pas par CALL, mais par JMP. Il y a une explication simple à cela: l'appelant pousse une valeur sur la pile et appelle notre fonction via CALL. CALL lui même pousse l'adresse de retour (RA) sur la pile et fait un saut inconditionnel à l'adresse de notre fonction. Notre fonction, à tout moment de son exécution (car elle ne contient pas d'instruction qui modifie le pointeur de pile) à le schéma suivant pour sa pile:

- ESP—pointe sur RA
- ESP+4—pointe sur la variable a

D'un autre côté, lorsque nous appelons `printf()` ici nous avons exactement la même disposition de pile, excepté pour le premier argument de `printf()`, qui doit pointer sur la chaîne. Et c'est ce que fait notre code.

Il remplace le premier argument de la fonction par l'adresse de la chaîne et saute à `printf()`, comme si nous n'avions pas appelé notre fonction `f()`, mais directement `printf()`. `printf()` affiche la chaîne sur la sortie standard et ensuite exécute l'instruction RET qui POPs RA de la pile et l'exécution est renvoyée non pas à `f()` mais plutôt à l'appelant de `f()`, ignorant la fin de la fonction `f()`.

Tout ceci est possible car `printf()` est appelée, dans tous les cas, tout à la fin de la fonction `f()`. Dans un certain sens, c'est similaire à la fonction `longjmp()`[94]. Et bien sûr, c'est fait dans un but de vitesse d'exécution.

Un cas similaire avec le compilateur ARM est décrit dans la section «printf() avec plusieurs arguments», ici (1.11.2 on page 75).

[94] Wikipédia

OllyDbg

Comme cet exemple est compliqué, traçons-le dans OllyDbg.

OllyDbg peut détecter des constructions avec switch(), et ajoute des commentaires utiles. EAX contient 2 au début, c'est la valeur du paramètre de la fonction:

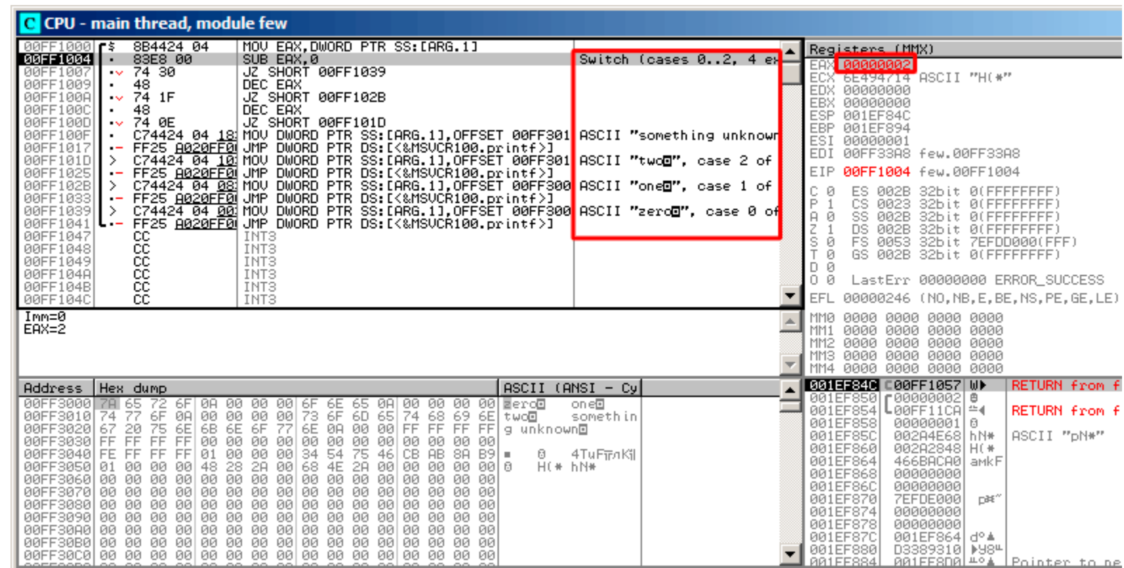


Fig. 1.42: OllyDbg : EAX contient maintenant le premier (et unique) argument de la fonction

0 est soustrait de 2 dans EAX. Bien sûr, EAX contient toujours 2. Mais le flag ZF est maintenant à 0, indiquant que le résultat est différent de zéro:

Fig. 1.43: OllyDbg : SUB exécuté

DEC est exécuté et EAX contient maintenant 1. Mais 1 est différent de zéro, donc le flag ZF est toujours à 0:

Fig. 1.44: OllyDbg : premier DEC exécuté

Le DEC suivant est exécuté. EAX contient maintenant 0 et le flag ZF est mis, car le résultat devient zéro:

Fig. 1.45: OllyDbg : second DEC exécuté

OllyDbg montre que le saut va être effectué (*Jump is taken*).

Un pointeur sur la chaîne «two» est maintenant écrit sur la pile:

Fig. 1.46: OllyDbg : pointeur sur la chaîne qui va être écrite à la place du premier argument

Veuillez noter: l'argument de la fonction courante est 2, et 2 est maintenant sur la pile, à l'adresse 0x001EF850.

MOV écrit le pointeur sur la chaîne à l'adresse 0x001EF850 (voir la fenêtre de la pile).
Puis, le saut est effectué. Ceci est la première instruction de la fonction printf() dans MSVCR100.DLL (Cet exemple a été compilé avec le switch /MD) :

Fig. 1.47: OllyDbg : première instruction de printf() dans MSVCR100.DLL

Maintenant printf() traite la chaîne à l'adresse 0x00FF3010 comme c'est son seul argument et l'affiche.

Ceci est la dernière instruction de `printf()` :

Fig. 1.48: OllyDbg : dernière instruction de `printf()` dans MSVCR100.DLL

La chaîne «two » vient juste d'être affichée dans la fenêtre console.

Maintenant, appuyez sur F7 ou F8 (enjamber) et le retour ne se fait pas sur f(), mais sur main() :

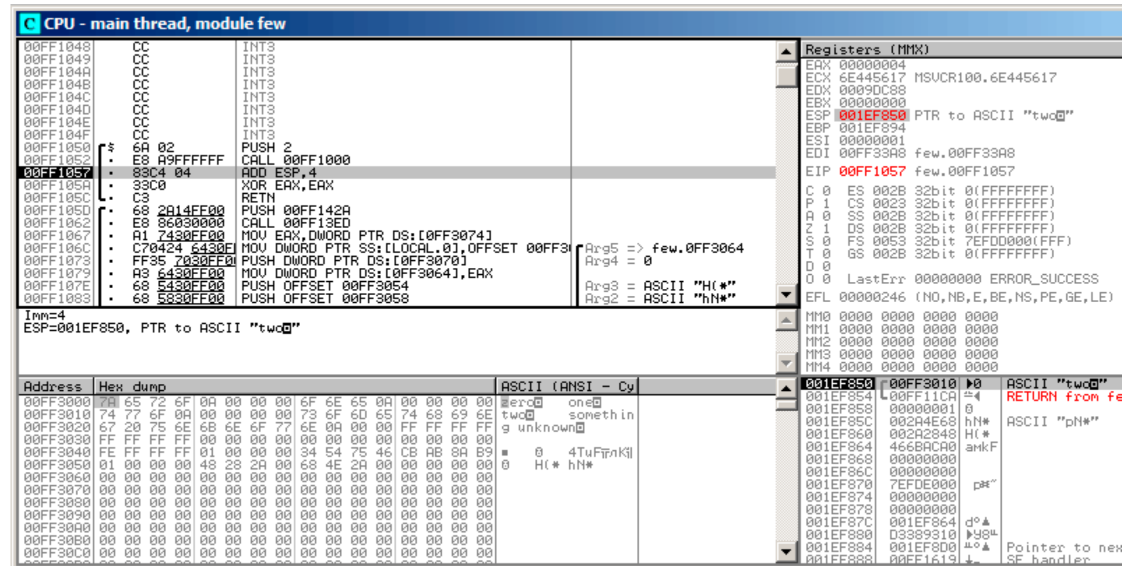


Fig. 1.49: OllyDbg : retourne à main()

Oui, le saut a été direct, depuis les entrailles de printf() vers main(). Car RA dans la pile pointe non pas quelque part dans f(), mais en fait sur main(). Et CALL 0x00FF1000 a été l'instruction qui a appelé f().

ARM: avec optimisation Keil 6/2013 (Mode ARM)

```
.text:0000014C             f1:
.text:0000014C 00 00 50 E3   CMP     R0, #0
.text:00000150 13 0E 8F 02   ADREQ   R0, aZero ; "zero\n"
.text:00000154 05 00 00 0A   BEQ     loc_170
.text:00000158 01 00 50 E3   CMP     R0, #1
.text:0000015C 4B 0F 8F 02   ADREQ   R0, aOne  ; "one\n"
.text:00000160 02 00 00 0A   BEQ     loc_170
.text:00000164 02 00 50 E3   CMP     R0, #2
.text:00000168 4A 0F 8F 12   ADRNE   R0, aSomethingUnkno ; "something unknown\n"
.text:0000016C 4E 0F 8F 02   ADREQ   R0, aTwo  ; "two\n"
.text:00000170
.text:00000170             loc_170: ; CODE XREF: f1+8
.text:00000170                      ; f1+14
.text:00000170 78 18 00 EA   B       __2printf
```

A nouveau, en investiguant ce code, nous ne pouvons pas dire si il y avait un switch() dans le code source d'origine ou juste un ensemble de déclarations if().

En tout cas, nous voyons ici des instructions conditionnelles (comme ADREQ (*Equal*)) qui ne sont exécutées que si $R0 = 0$, et qui chargent ensuite l'adresse de la chaîne «*zero\n*» dans R0. L'instruction suivante BEQ redirige le flux d'exécution en loc_170, si $R0 = 0$.

Le lecteur attentif peut se demander si BEQ s'exécute correctement puisque ADREQ a déjà mis une autre valeur dans le registre R0.

Oui, elle s'exécutera correctement, car BEQ vérifie les flags mis par l'instruction CMP et ADREQ ne modifie aucun flag.

Les instructions restantes nous sont déjà familières. Il y a seulement un appel à printf(), à la fin, et nous avons déjà examiné cette astuce ici (1.11.2 on page 75). A la fin, il y a trois chemins vers printf().

La dernière instruction, CMP R0, #2, est nécessaire pour vérifier si $a = 2$.

Si ce n'est pas vrai, alors ADRNE charge un pointeur sur la chaîne «*something unknown \n*» dans R0, puisque a a déjà été comparée pour savoir s'elle est égale à 0 ou 1, et nous sommes sûrs que la variable a n'est pas égale à l'un de ces nombres, à ce point. Et si $R0 = 2$, un pointeur sur la chaîne «*two\n*» sera chargé par ADREQ dans R0.

ARM: avec optimisation Keil 6/2013 (Mode Thumb)

```
.text:000000D4             f1:
.text:000000D4 10 B5       PUSH    {R4,LR}
.text:000000D6 00 28       CMP     R0, #0
.text:000000D8 05 D0       BEQ     zero_case
.text:000000DA 01 28       CMP     R0, #1
.text:000000DC 05 D0       BEQ     one_case
.text:000000DE 02 28       CMP     R0, #2
.text:000000E0 05 D0       BEQ     two_case
.text:000000E2 91 A0       ADR     R0, aSomethingUnkno ; "something
    unknown\n"
.text:000000E4 04 E0       B       default_case

.text:000000E6             zero_case: ; CODE XREF: f1+4
.text:000000E6 95 A0       ADR     R0, aZero ; "zero\n"
.text:000000E8 02 E0       B       default_case

.text:000000EA             one_case: ; CODE XREF: f1+8
.text:000000EA 96 A0       ADR     R0, aOne ; "one\n"
.text:000000EC 00 E0       B       default_case

.text:000000EE             two_case: ; CODE XREF: f1+C
.text:000000EE 97 A0       ADR     R0, aTwo ; "two\n"
.text:000000F0             default_case ; CODE XREF: f1+10
.text:000000F0                                     ; f1+14
.text:000000F0 06 F0 7E F8 BL      __2printf
.text:000000F4 10 BD       POP     {R4,PC}
```

Comme il y déjà été dit, il n'est pas possible d'ajouter un prédicat conditionnel à la plupart des instructions en mode Thumb, donc ce dernier est quelque peu similaire

au code CISC-style x86, facilement compréhensible.

ARM64: GCC (Linaro) 4.9 sans optimisation

```
.LC12:
        .string "zero"
.LC13:
        .string "one"
.LC14:
        .string "two"
.LC15:
        .string "something unknown"
f12:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        str     w0, [x29,28]
        ldr     w0, [x29,28]
        cmp     w0, 1
        beq     .L34
        cmp     w0, 2
        beq     .L35
        cmp     w0, wzr
        bne     .L38            ; sauter au label par défaut
        adrp    x0, .LC12       ; "zero"
        add     x0, x0, :lo12:.LC12
        bl      puts
        b       .L32
.L34:
        adrp    x0, .LC13       ; "one"
        add     x0, x0, :lo12:.LC13
        bl      puts
        b       .L32
.L35:
        adrp    x0, .LC14       ; "two"
        add     x0, x0, :lo12:.LC14
        bl      puts
        b       .L32
.L38:
        adrp    x0, .LC15       ; "something unknown"
        add     x0, x0, :lo12:.LC15
        bl      puts
        nop
.L32:
        ldp     x29, x30, [sp], 32
        ret
```

Le type de la valeur d'entrée est *int*, par conséquent le registre `W0` est utilisé pour garder la valeur au lieu du registre complet `X0`.

Les pointeurs de chaîne sont passés à `puts()` en utilisant la paire d'instructions `ADRP`/`ADD` comme expliqué dans l'exemple «Hello, world! » : 1.5.3 on page 33.

ARM64: GCC (Linaro) 4.9 avec optimisation

```
f12:
        cmp     w0, 1
        beq     .L31
        cmp     w0, 2
        beq     .L32
        cbz     w0, .L35
; cas par défaut
        adrp    x0, .LC15       ; "something unknown"
        add     x0, x0, :lo12:.LC15
        b       puts
.L35:
        adrp    x0, .LC12       ; "zero"
        add     x0, x0, :lo12:.LC12
        b       puts
.L32:
        adrp    x0, .LC14       ; "two"
        add     x0, x0, :lo12:.LC14
        b       puts
.L31:
        adrp    x0, .LC13       ; "one"
        add     x0, x0, :lo12:.LC13
        b       puts
```

Ce morceau de code est mieux optimisé. L'instruction CBZ (*Compare and Branch on Zero* comparer et sauter si zéro) effectue un saut si W0 vaut zéro. Il y a alors un saut direct à puts() au lieu de l'appeler, comme cela a été expliqué avant: 1.21.1 on page 205.

MIPS

Listing 1.156 : GCC 4.4.5 avec optimisation (IDA)

```
f:
                lui     $gp, (__gnu_local_gp >> 16)
; est-ce 1?
                li      $v0, 1
                beq     $a0, $v0, loc_60
                la      $gp, (__gnu_local_gp & 0xFFFF) ; slot de délai de
    branchement
; est-ce 2?
                li      $v0, 2
                beq     $a0, $v0, loc_4C
                or      $at, $zero ; slot de délai de branchement, NOP
; jump, if not equal to 0:
                bnez    $a0, loc_38
                or      $at, $zero ; slot de délai de branchement, NOP
; cas zéro:
                lui     $a0, ($LC0 >> 16)  # "zero"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; slot de délai de branchement, NOP
                jr      $t9
```

```
                la      $a0, ($LC0 & 0xFFFF)  # "zero" ; slot de délai de
    branchement

loc_38:                                         # CODE XREF: f+1C
                lui     $a0, ($LC3 >> 16)  # "something unknown"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; slot de délai de branchement, NOP
                jr      $t9
                la      $a0, ($LC3 & 0xFFFF)  # "something unknown" ; slot de
    délai de branchement

loc_4C:                                         # CODE XREF: f+14
                lui     $a0, ($LC2 >> 16)  # "two"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; slot de délai de branchement, NOP
                jr      $t9
                la      $a0, ($LC2 & 0xFFFF)  # "two" ; slot de délai de
    branchement

loc_60:                                         # CODE XREF: f+8
                lui     $a0, ($LC1 >> 16)  # "one"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; slot de délai de branchement, NOP
                jr      $t9
                la      $a0, ($LC1 & 0xFFFF)  # "one" ; slot de délai de
    branchement
```

La fonction se termine toujours en appelant `puts()`, donc nous voyons un saut à `puts()` (JR : «Jump Register») au lieu de «jump and link». Nous avons parlé de ceci avant: 1.21.1 on page 205.

Nous voyons aussi souvent l'instruction `NOP` après `LW`. Ceci est le slot de délai de chargement («load delay slot») : un autre slot de délai (*delay slot*) en MIPS.

Une instruction suivant `LW` peut s'exécuter pendant que `LW` charge une valeur depuis la mémoire.

Toutefois, l'instruction suivante ne doit pas utiliser le résultat de `LW`.

Les CPU MIPS modernes ont la capacité d'attendre si l'instruction suivante utilise le résultat de `LW`, donc ceci est un peu démodé, mais GCC ajoute toujours des NOPs pour les anciens CPU MIPS. En général, ça peut être ignoré.

Conclusion

Un *switch()* avec peu de cas est indistinguable d'une construction avec *if/else*, par exemple: listado.1.21.1.

1.21.2 De nombreux cas

Si une déclaration `switch()` contient beaucoup de cas, il n'est pas très pratique pour le compilateur de générer un trop gros code avec de nombreuses instructions `JE`/`JNE`.

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    case 3: printf ("three\n"); break;
    case 4: printf ("four\n"); break;
    default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};
```

x86

MSVC sans optimisation

Nous obtenons (MSVC 2010) :

Listing 1.157 : MSVC 2010

```
tv64 = -4  ; size = 4
_a$ = 8    ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 4
    ja     SHORT $LN1@f
    mov    ecx, DWORD PTR tv64[ebp]
    jmp    DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN5@f:
    push   OFFSET $SG741 ; 'one', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN4@f:
    push   OFFSET $SG743 ; 'two', 0aH, 00H
```

```
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN3@f:
    push   OFFSET $SG745 ; 'three', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN2@f:
    push   OFFSET $SG747 ; 'four', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN1@f:
    push   OFFSET $SG749 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN9@f:
    mov    esp, ebp
    pop    ebp
    ret    0
    npad    2 ; aligner le label suivant
$LN11@f:
    DD    $LN6@f ; 0
    DD    $LN5@f ; 1
    DD    $LN4@f ; 2
    DD    $LN3@f ; 3
    DD    $LN2@f ; 4
_f    ENDP
```

Ce que nous voyons ici est un ensemble d'appels à `printf()` avec des arguments variés. Ils ont tous, non seulement des adresses dans la mémoire du processus, mais aussi des labels symboliques internes assignés par le compilateur. Tous ces labels ont aussi mentionnés dans la table interne `$LN11@f`.

Au début de la fonctions, si a est supérieur à 4, l'exécution est passée au labal `$LN1@f`, où `printf()` est appelé avec l'argument `'something unknown'`.

Mais si la valeur de a est inférieure ou égale à 4, elle est alors multipliée par 4 et ajoutée à l'adresse de la table `$LN11@f`. C'est ainsi qu'une adresse à l'intérieur de la table est construite, pointant exactement sur l'élément dont nous avons besoin. Par exemple, supposons que a soit égale à 2. $2 * 4 = 8$ (tous les éléments de la table sont adressés dans un processus 32-bit, c'est pourquoi les éléments ont une taille de 4 octets). L'adresse de la table `$LN11@f` + 8 est celle de l'élément de la table où le label `$LN4@f` est stocké. JMP prend l'adresse de `$LN4@f` dans la table et y saute.

Cette table est quelquefois appelée *jumptable* (table de saut) ou *branch table* (table de branchement)[95].

Le `printf()` correspondant est appelé avec l'argument `'two'`.
Littéralement, l'instruction `jmp DWORD PTR $LN11@f[ecx*4]` signifie *sauter au DWORD*

[95]L'ensemble de la méthode était appelé *computed GOTO* (GOTO calculés) dans les premières versions de ForTran: Wikipédia. Pas très pertinent de nos jours, mais quel terme!

qui est stocké à l'adresse `$LN11@f + ecx * 4`.

npad (.1.7 on page 1347) est une macro du langage d'assemblage qui aligne le label suivant de telle sorte qu'il soit stocké à une adresse alignée sur une limite de 4 octets (ou 16 octets). C'est très adapté pour le processeur puisqu'il est capable d'aller chercher des valeurs 32-bit dans la mémoire à travers le bus mémoire, la mémoire cache, etc., de façons beaucoup plus efficace si c'est aligné.

OllyDbg

Essayons cet exemple dans OllyDbg. La valeur d'entrée de la fonction (2) est chargée dans EAX :

Fig. 1.50: OllyDbg : la valeur d'entrée de la fonction est chargée dans EAX

La valeur entrée est testée, est-elle plus grande que 4? Si non, le saut par «défaut» n’est pas pris:

Fig. 1.51: OllyDbg : 2 n’est pas plus grand que 4: le saut n’est pas pris

Ici, nous voyons une table des sauts:

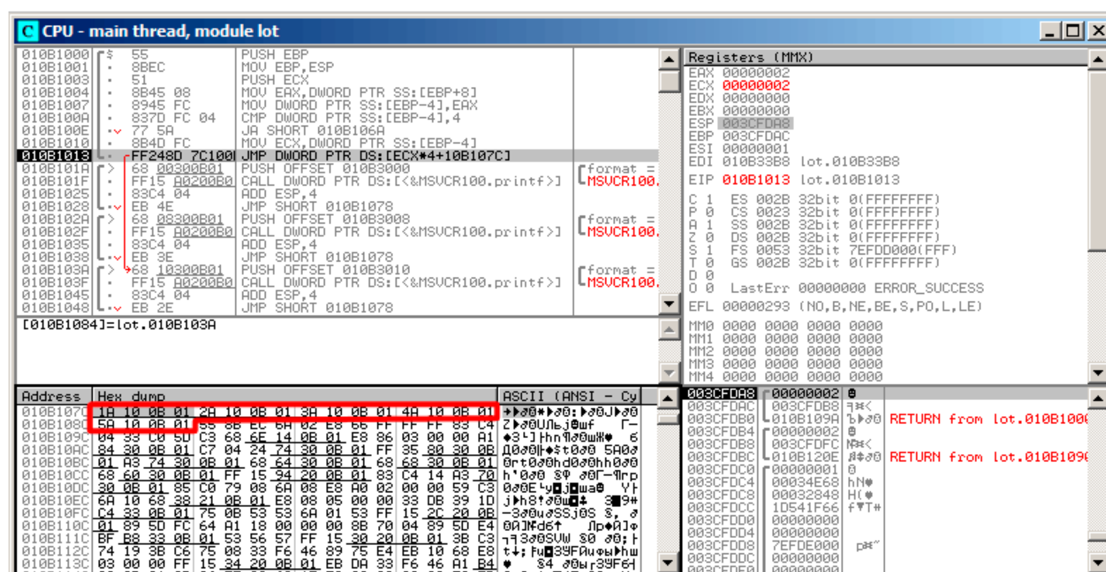


Fig. 1.52: OllyDbg : calcul de l'adresse de destination en utilisant la table des sauts

Ici, nous avons cliqué «Follow in Dump» → «Address constant», donc nous voyons maintenant la *jumptable* dans la fenêtre des données. Il y a 5 valeurs 32-bit[96]. ECX contient maintenant 2, donc le troisième élément (peut être indexé par 2[97]) de la table va être utilisé. Il est également possible de cliquer sur «Follow in Dump» → «Memory address» et OllyDbg va montrer l'élément adressé par l'instruction JMP. Il s'agit de `0x010B103A`.

[96]Elles sont soulignées par OllyDbg car ce sont aussi des FIXUPs: 6.5.2 on page 989, nous y reviendrons plus tard

[97]À propos des index de tableaux, lire aussi: 3.22.3 on page 770

Après le saut, nous sommes en 0x010B103A : le code qui affiche «two» va être exécuté:

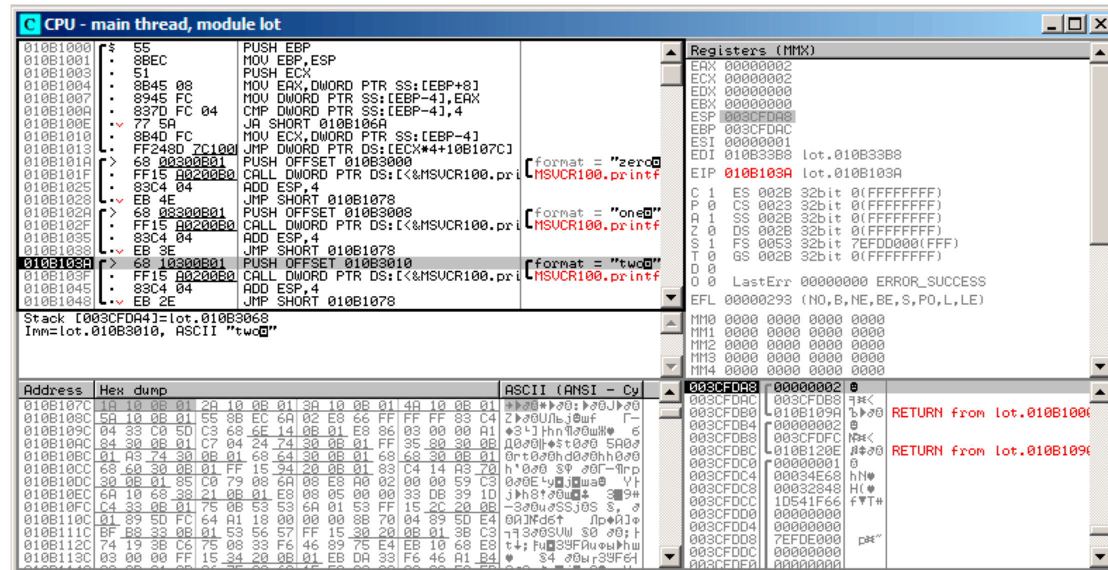


Fig. 1.53: OllyDbg : maintenant nous sommes au *cas:* label

GCC sans optimisation

Voyons ce que GCC 4.4.1 génère:

Listing 1.158 : GCC 4.4.1

```
        public f
f       proc near ; CODE XREF: main+10

var_18 = dword ptr -18h
arg_0  = dword ptr  8

        push    ebp
        mov     ebp, esp
        sub     esp, 18h
        cmp     [ebp+arg_0], 4
        ja      short loc_8048444
        mov     eax, [ebp+arg_0]
        shl     eax, 2
        mov     eax, ds:off_804855C[eax]
        jmp     eax

loc_80483FE: ; DATA XREF: .rodata:off_804855C
        mov     [esp+18h+var_18], offset aZero ; "zero"
        call    _puts
        jmp     short locret_8048450
```

```
loc_804840C: ; DATA XREF: .rodata:08048560
       mov     [esp+18h+var_18], offset aOne ; "one"
       call    _puts
       jmp     short locret_8048450

loc_804841A: ; DATA XREF: .rodata:08048564
       mov     [esp+18h+var_18], offset aTwo ; "two"
       call    _puts
       jmp     short locret_8048450

loc_8048428: ; DATA XREF: .rodata:08048568
       mov     [esp+18h+var_18], offset aThree ; "three"
       call    _puts
       jmp     short locret_8048450

loc_8048436: ; DATA XREF: .rodata:0804856C
       mov     [esp+18h+var_18], offset aFour ; "four"
       call    _puts
       jmp     short locret_8048450

loc_8048444: ; CODE XREF: f+A
       mov     [esp+18h+var_18], offset aSomethingUnkno ; "something unknown"
       call    _puts

locret_8048450: ; CODE XREF: f+26
                ; f+34...
       leave
       retn
f      endp

off_804855C dd offset loc_80483FE   ; DATA XREF: f+12
            dd offset loc_804840C
            dd offset loc_804841A
            dd offset loc_8048428
            dd offset loc_8048436
```

C'est presque la même chose, avec une petite nuance: l'argument `arg_0` est multiplié par 4 en décalant de 2 bits vers la gauche (c'est presque comme multiplier par 4) (1.24.2 on page 284). Ensuite l'adresse du label est prise depuis le tableau `off_804855C`, stockée dans EAX, et ensuite `JMP EAX` effectue le saut réel.

ARM: avec optimisation Keil 6/2013 (Mode ARM)

Listing 1.159 : avec optimisation Keil 6/2013 (Mode ARM)

```
00000174             f2
00000174 05 00 50 E3     CMP     R0, #5           ; switch 5 cases
00000178 00 F1 8F 30     ADDCC   PC, PC, R0,LSL#2 ; switch jump
0000017C 0E 00 00 EA     B       default_case     ; jumptable 00000178
    default case

00000180
```

```
00000180             loc_180 ; CODE XREF: f2+4
00000180 03 00 00 EA     B       zero_case       ; jumptable 00000178 case 0

00000184
00000184             loc_184 ; CODE XREF: f2+4
00000184 04 00 00 EA     B       one_case        ; jumptable 00000178 case 1

00000188
00000188             loc_188 ; CODE XREF: f2+4
00000188 05 00 00 EA     B       two_case        ; jumptable 00000178 case 2

0000018C
0000018C             loc_18C ; CODE XREF: f2+4
0000018C 06 00 00 EA     B       three_case      ; jumptable 00000178 case 3

00000190
00000190             loc_190 ; CODE XREF: f2+4
00000190 07 00 00 EA     B       four_case       ; jumptable 00000178 case 4

00000194
00000194             zero_case ; CODE XREF: f2+4
00000194                       ; f2:loc_180
00000194 EC 00 8F E2     ADR     R0, aZero       ; jumptable 00000178 case 0
00000198 06 00 00 EA     B       loc_1B8

0000019C
0000019C             one_case ; CODE XREF: f2+4
0000019C                      ; f2:loc_184
0000019C EC 00 8F E2     ADR     R0, aOne        ; jumptable 00000178 case 1
000001A0 04 00 00 EA     B       loc_1B8

000001A4
000001A4             two_case ; CODE XREF: f2+4
000001A4                      ; f2:loc_188
000001A4 01 0C 8F E2     ADR     R0, aTwo        ; jumptable 00000178 case 2
000001A8 02 00 00 EA     B       loc_1B8

000001AC
000001AC             three_case ; CODE XREF: f2+4
000001AC                        ; f2:loc_18C
000001AC 01 0C 8F E2     ADR     R0, aThree      ; jumptable 00000178 case 3
000001B0 00 00 00 EA     B       loc_1B8

000001B4
000001B4             four_case ; CODE XREF: f2+4
000001B4                       ; f2:loc_190
000001B4 01 0C 8F E2     ADR     R0, aFour       ; jumptable 00000178 case 4
000001B8
000001B8             loc_1B8   ; CODE XREF: f2+24
000001B8                       ; f2+2C
000001B8 66 18 00 EA     B       __2printf

000001BC
```

```
000001BC              default_case ; CODE XREF: f2+4
000001BC                           ; f2+8
000001BC D4 00 8F E2      ADR     R0, aSomethingUnkno ; jumptable 00000178
    default case
000001C0 FC FF FF EA      B       loc_1B8
```

Ce code utilise les caractéristiques du mode ARM dans lequel toutes les instructions ont une taille fixe de 4 octets.

Gardons à l'esprit que la valeur maximale de a est 4 et que toute autre valeur supérieure provoquera l'affichage de la chaîne *«something unknown\n»*

La première instruction CMP R0, #5 compare la valeur entrée dans a avec 5.

[98] L'instruction suivante, ADDCC PC, PC, R0,LSL#2, est exécutée si et seulement si $R0 < 5$ (*CC=Carry clear / Less than* retenue vide, inférieur à). Par conséquent, si ADDCC n'est pas exécutée (c'est le cas $R0 \geq 5$), un saut au label *default_case* se produit.

Mais si $R0 < 5$ et que ADDCC est exécuté, voici ce qui se produit:

La valeur dans R0 est multipliée par 4. En fait, le suffixe de l'instruction LSL#2 signifie «décalage à gauche de 2 bits». Mais comme nous le verrons plus tard (1.24.2 on page 284) dans la section «Décalages», décaler de 2 bits vers la gauche est équivalent à multiplier par 4.

Puis, nous ajoutons $R0 * 4$ à la valeur courante du PC, et sautons à l'une des instructions B (*Branch*) situées plus bas.

Au moment de l'exécution de ADDCC, la valeur du PC est en avance de 8 octets (0x180) sur l'adresse à laquelle l'instruction ADDCC se trouve (0x178), ou, autrement dit, en avance de 2 instructions.

C'est ainsi que le pipeline des processeurs ARM fonctionne: lorsque ADDCC est exécutée, le processeur, à ce moment, commence à préparer les instructions après la suivante, c'est pourquoi PC pointe ici. Cela doit être mémorisé.

Si $a = 0$, elle sera ajoutée à la valeur de PC, et la valeur courante de PC sera écrite dans PC (qui est 8 octets en avant) et un saut au label *loc_180* sera effectué, qui est 8 octets en avant du point où l'instruction se trouve.

Si $a = 1$, alors $PC + 8 + a * 4 = PC + 8 + 1 * 4 = PC + 12 = 0x184$ sera écrit dans PC, qui est l'adresse du label *loc_184*.

A chaque fois que l'on ajoute 1 à a, le PC résultant est incrémenté de 4.

4 est la taille des instructions en mode ARM, et donc, la longueur de chaque instruction B desquelles il y a 5 à la suite.

Chacune de ces cinq instructions B passe le contrôle plus loin, à ce qui a été programmé dans le *switch()*.

Le chargement du pointeur sur la chaîne correspondante se produit ici, etc.

[98]ADD—addition

ARM: avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.160 : avec optimisation Keil 6/2013 (Mode Thumb)

```
000000F6                          EXPORT f2
000000F6                      f2
000000F6 10 B5                    PUSH    {R4,LR}
000000F8 03 00                    MOVS    R3, R0
000000FA 06 F0 69 F8              BL      __ARM_common_switch8_thumb ; switch 6
    cases

000000FE 05                       DCB 5
000000FF 04 06 08 0A 0C 10        DCB 4, 6, 8, 0xA, 0xC, 0x10 ; jump table for
    switch statement
00000105 00                       ALIGN 2
00000106
00000106                      zero_case ; CODE XREF: f2+4
00000106 8D A0                    ADR     R0, aZero ; jumptable 000000FA case 0
00000108 06 E0                    B       loc_118

0000010A
0000010A                      one_case ; CODE XREF: f2+4
0000010A 8E A0                    ADR     R0, aOne ; jumptable 000000FA case 1
0000010C 04 E0                    B       loc_118

0000010E
0000010E                      two_case ; CODE XREF: f2+4
0000010E 8F A0                    ADR     R0, aTwo ; jumptable 000000FA case 2
00000110 02 E0                    B       loc_118

00000112
00000112                      three_case ; CODE XREF: f2+4
00000112 90 A0                    ADR     R0, aThree ; jumptable 000000FA case 3
00000114 00 E0                    B       loc_118

00000116
00000116                      four_case ; CODE XREF: f2+4
00000116 91 A0                    ADR     R0, aFour ; jumptable 000000FA case 4
00000118
00000118                      loc_118 ; CODE XREF: f2+12
00000118                              ; f2+16
00000118 06 F0 6A F8              BL      __2printf
0000011C 10 BD                    POP     {R4,PC}

0000011E
0000011E                      default_case ; CODE XREF: f2+4
0000011E 82 A0                    ADR     R0, aSomethingUnkno ; jumptable
    000000FA default case
00000120 FA E7                    B       loc_118


000061D0                          EXPORT __ARM_common_switch8_thumb
000061D0                      __ARM_common_switch8_thumb ; CODE XREF:
    example6_f2+4
000061D0 78 47                    BX      PC

```

```
000061D2 00 00                  ALIGN 4
000061D2                    ; End of function __ARM_common_switch8_thumb
000061D2
000061D4                    __32__ARM_common_switch8_thumb ; CODE XREF:
    __ARM_common_switch8_thumb
000061D4 01 C0 5E E5            LDRB    R12, [LR,#-1]
000061D8 0C 00 53 E1            CMP     R3, R12
000061DC 0C 30 DE 27            LDRCSB  R3, [LR,R12]
000061E0 03 30 DE 37            LDRCCB  R3, [LR,R3]
000061E4 83 C0 8E E0            ADD     R12, LR, R3,LSL#1
000061E8 1C FF 2F E1            BX      R12
000061E8                    ; End of function __32__ARM_common_switch8_thumb
```

On ne peut pas être sûr que toutes ces instructions en mode Thumb et Thumb-2 ont la même taille. On peut même dire que les instructions dans ces modes ont une longueur variable, tout comme en x86.

Donc, une table spéciale est ajoutée, qui contient des informations sur le nombre de cas (sans inclure celui par défaut), et un offset pour chaque label auquel le contrôle doit être passé dans chaque cas.

Une fonction spéciale est présente ici qui s'occupe de la table et du passage du contrôle, appelée
__ARM_common_switch8_thumb. Elle commence avec `BX PC`, dont la fonction est de passer le mode du processeur en ARM. Ensuite, vous voyez la fonction pour le traitement de la table.

C'est trop avancé pour être détaillé ici, donc passons cela.

Il est intéressant de noter que la fonction utilise le registre LR comme un pointeur sur la table.

En effet, après l'appel de cette fonction, LR contient l'adresse après l'instruction `BL __ARM_common_switch8_thumb`, où la table commence.

Il est intéressant de noter que le code est généré comme une fonction indépendante afin de la ré-utiliser, donc le compilateur ne générera pas le même code pour chaque déclaration switch().

IDA l'a correctement identifié comme une fonction de service et une table, et a ajouté un commentaire au label comme `jumptable 000000FA case 0`.

MIPS

Listing 1.161 : GCC 4.4.5 avec optimisation (IDA)

```
f:
                lui     $gp, (__gnu_local_gp >> 16)
; sauter en loc_24 si la valeur entrée est plus petite que 5:
                sltiu   $v0, $a0, 5
                bnez    $v0, loc_24
                la      $gp, (__gnu_local_gp & 0xFFFF) ; slot de délai de
    branchement
; la valeur entrée est supérieur ou égale à 5.
; afficher "something unknown" et terminer:
```

```
                lui     $a0, ($LC5 >> 16)  # "something unknown"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC5 & 0xFFFF)  # "something unknown"
                                              ; slot de délai de branchement

loc_24:                                   # CODE XREF: f+8
; charger l'adresse de la table de branchement/saut
; LA est une pseudo instruction, il s'agit en fait de LUI et ADDIU:
                la      $v0, off_120
; multiplier la valeur entrés par 4:
                sll     $a0, 2
; ajouter la valeur multipliée et l'adresse de la table de saut:
                addu    $a0, $v0, $a0
; charger l'élément de la table de saut:
                lw      $v0, 0($a0)
                or      $at, $zero ; NOP
; sauter à l'adresse que nous avons dans la table de saut:
                jr      $v0
                or      $at, $zero ; slot de délai de branchement, NOP

sub_44:                                   # DATA XREF: .rodata:0000012C
; afficher "three" et terminer
                lui     $a0, ($LC3 >> 16)  # "three"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC3 & 0xFFFF)  # "three" ; slot de délai de
    branchement

sub_58:                                   # DATA XREF: .rodata:00000130
; afficher "four" et terminer
                lui     $a0, ($LC4 >> 16)  # "four"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC4 & 0xFFFF)  # "four" ; slot de délai de
    branchement

sub_6C:                                   # DATA XREF: .rodata:off_120
; afficher "zero" et terminer
                lui     $a0, ($LC0 >> 16)  # "zero"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC0 & 0xFFFF)  # "zero" ; slot de délai de
    branchement

sub_80:                                   # DATA XREF: .rodata:00000124
; afficher "one" et terminer
                lui     $a0, ($LC1 >> 16)  # "one"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
```

```
                jr      $t9
                la      $a0, ($LC1 & 0xFFFF)  # "one" ; slot de délai de
    branchement

sub_94:                                  # DATA XREF: .rodata:00000128
; afficher "two" et terminer
                lui     $a0, ($LC2 >> 16)  # "two"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC2 & 0xFFFF)  # "two" ; slot de délai de
    branchement

; peut être mis dans une section .rodata:
off_120:        .word sub_6C
                .word sub_80
                .word sub_94
                .word sub_44
                .word sub_58
```

La nouvelle instruction pour nous est SLTIU («Set on Less Than Immediate Unsigned» Mettre si inférieur à la valeur immédiate non signée).

Ceci est la même que SLTU («Set on Less Than Unsigned»), mais «I» signifie «immediate», i.e., un nombre doit être spécifié dans l'instruction elle-même.

BNEZ est «Branch if Not Equal to Zero».

Le code est très proche de l'autre ISAs. SLL («Shift Word Left Logical») effectue une multiplication par 4.

MIPS est un CPU 32-bit après tout, donc toutes les adresses de la *jumtable* sont 32-bits.

Conclusion

Squelette grossier d'un *switch()* :

Listing 1.162 : x86

```
MOV REG, input
CMP REG, 4 ; nombre maximal de cas
JA default
SHL REG, 2 ; trouver l'élément dans la table. décaler de 3 bits en x64.
MOV REG, jump_table[REG]
JMP REG

case1:
    ; faire quelque chose
    JMP exit
case2:
    ; faire quelque chose
    JMP exit
case3:
    ; faire quelque chose
```

```
    JMP exit
case4:
    ; faire quelque chose
    JMP exit
case5:
    ; faire quelque chose
    JMP exit

default:

    ...

exit:

    ....

jump_table dd case1
           dd case2
           dd case3
           dd case4
           dd case5
```

Le saut à une adresse de la table de saut peut aussi être implémenté en utilisant cette instruction:
`JMP jump_table[REG*4]`. Ou `JMP jump_table[REG*8]` en x64.

Une table de saut est juste un tableau de pointeurs, comme celle décrite plus loin: 1.26.5 on page 364.

1.21.3 Lorsqu'il y a quelques déclarations *case* dans un bloc

Voici une construction très répandue: quelques déclarations *case* pour un seul bloc:

```
#include <stdio.h>

void f(int a)
{
        switch (a)
        {
        case 1:
        case 2:
        case 7:
        case 10:
                printf ("1, 2, 7, 10\n");
                break;
        case 3:
        case 4:
        case 5:
        case 6:
                printf ("3, 4, 5\n");
                break;
        case 8:
        case 9:
```

```
        case 20:
        case 21:
                printf ("8, 9, 21\n");
                break;
        case 22:
                printf ("22\n");
                break;
        default:
                printf ("default\n");
                break;
        };
};

int main()
{
        f(4);
};
```

C'est souvent du gaspillage de générer un bloc pour chaque cas possible, c'est pourquoi ce qui se fait d'habitude, c'est de générer un bloc et une sorte de répartiteur.

MSVC

Listing 1.163 : MSVC 2010 avec optimisation

```
$SG2798 DB      '1, 2, 7, 10', 0aH, 00H
$SG2800 DB      '3, 4, 5', 0aH, 00H
$SG2802 DB      '8, 9, 21', 0aH, 00H
$SG2804 DB      '22', 0aH, 00H
$SG2806 DB      'default', 0aH, 00H

_a$ = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        dec     eax
        cmp     eax, 21
        ja      SHORT $LN1@f
        movzx   eax, BYTE PTR $LN10@f[eax]
        jmp     DWORD PTR $LN11@f[eax*4]
$LN5@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2798 ; '1, 2, 7, 10'
        jmp     DWORD PTR __imp__printf
$LN4@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2800 ; '3, 4, 5'
        jmp     DWORD PTR __imp__printf
$LN3@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2802 ; '8, 9, 21'
        jmp     DWORD PTR __imp__printf
$LN2@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2804 ; '22'
        jmp     DWORD PTR __imp__printf
$LN1@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2806 ; 'default'
```

```
29          jmp     DWORD PTR __imp__printf
30          npad    2 ; aligner la table $LN11@f sur une limite de 16-octet
31  $LN11@f:
32          DD      $LN5@f ; afficher '1, 2, 7, 10'
33          DD      $LN4@f ; afficher '3, 4, 5'
34          DD      $LN3@f ; afficher '8, 9, 21'
35          DD      $LN2@f ; afficher '22'
36          DD      $LN1@f ; afficher 'default'
37  $LN10@f:
38          DB      0 ; a=1
39          DB      0 ; a=2
40          DB      1 ; a=3
41          DB      1 ; a=4
42          DB      1 ; a=5
43          DB      1 ; a=6
44          DB      0 ; a=7
45          DB      2 ; a=8
46          DB      2 ; a=9
47          DB      0 ; a=10
48          DB      4 ; a=11
49          DB      4 ; a=12
50          DB      4 ; a=13
51          DB      4 ; a=14
52          DB      4 ; a=15
53          DB      4 ; a=16
54          DB      4 ; a=17
55          DB      4 ; a=18
56          DB      4 ; a=19
57          DB      2 ; a=20
58          DB      2 ; a=21
59          DB      3 ; a=22
60  _f      ENDP
```

Nous voyons deux tables ici: la première ($LN10@f) est une table d’index, et la seconde ($LN11@f) est un tableau de pointeurs sur les blocs.

Tout d’abord, la valeur entrée est utilisée comme un index dans la table d’index (ligne 13).

Voici un petit récapitulatif pour les valeurs dans la table: 0 est le premier bloc *case* (pour les valeurs 1, 2, 7, 10), 1 est le second (pour les valeurs 3, 4, 5), 2 est le troisième (pour les valeurs 8, 9, 21), 3 est le quatrième (pour la valeur 22), 4 est pour le bloc par défaut.

Ici, nous obtenons un index pour la seconde table de pointeurs sur du code et nous y sautons (ligne 14).

Il est intéressant de remarquer qu’il n’y a pas de cas pour une valeur d’entrée de 0.

C’est pourquoi nous voyons l’instruction DEC à la ligne 10, et la table commence à $a = 1$, car il n’y a pas besoin d’allouer un élément dans la table pour $a = 0$.

C’est un pattern très répandu.

Donc, pourquoi est-ce que c’est économique ? Pourquoi est-ce qu’il n’est pas possible

de faire comme avant (1.21.2 on page 226), avec une seule table consistant en des pointeurs vers les blocs? La raison est que les index des éléments de la table sont 8-bit, donc c'est plus compact.

GCC

GCC génère du code de la façon dont nous avons déjà discuté (1.21.2 on page 226), en utilisant juste une table de pointeurs.

ARM64: GCC 4.9.1 avec optimisation

Il n'y a pas de code à exécuter si la valeur entrée est 0, c'est pourquoi GCC essaye de rendre la table des sauts plus compacte et donc il commence avec la valeur d'entrée 1.

GCC 4.9.1 pour ARM64 utilise un truc encore plus astucieux. Il est capable d'encoder tous les offsets en octets 8-bit.

Rappelons-nous que toutes les instructions ARM64 ont une taille de 4 octets.

GCC utilise le fait que tous les offsets de mon petit exemple sont tous proche l'un de l'autre. Donc la table des sauts consiste en de simple octets.

Listing 1.164 : avec optimisation GCC 4.9.1 ARM64

```
f14:
; valeur entrée dans W0
        sub     w0, w0, #1
        cmp     w0, 21
; branchement si inférieur ou égal (non signé):
        bls     .L9
.L2:
; afficher "default":
        adrp    x0, .LC4
        add     x0, x0, :lo12:.LC4
        b       puts
.L9:
; charger l'adresse de la table des sauts dans X1:
        adrp    x1, .L4
        add     x1, x1, :lo12:.L4
; W0=input_value-1
; charger un octet depuis la table:
        ldrb    w0, [x1,w0,uxtw]
; charger l'adresse du label .Lrtx4:
        adr     x1, .Lrtx4
; multiplier l'élément de la table par 4 (en décalant de 2 bits à gauche) et
; ajouter (ou soustraire) à l'adresse de .Lrtx4:
        add     x0, x1, w0, sxtb #2
; sauter à l'adresse calculée:
        br      x0
; ce label pointe dans le segment de code (text):
.Lrtx4:
        .section        .rodata
```

```
; tout ce qui se trouve après la déclaration ".section" est alloué dans le
    segment de données
; en lecture seule (rodata):
.L4:
        .byte   (.L3 - .Lrtx4) / 4     ; case 1
        .byte   (.L3 - .Lrtx4) / 4     ; case 2
        .byte   (.L5 - .Lrtx4) / 4     ; case 3
        .byte   (.L5 - .Lrtx4) / 4     ; case 4
        .byte   (.L5 - .Lrtx4) / 4     ; case 5
        .byte   (.L5 - .Lrtx4) / 4     ; case 6
        .byte   (.L3 - .Lrtx4) / 4     ; case 7
        .byte   (.L6 - .Lrtx4) / 4     ; case 8
        .byte   (.L6 - .Lrtx4) / 4     ; case 9
        .byte   (.L3 - .Lrtx4) / 4     ; case 10
        .byte   (.L2 - .Lrtx4) / 4     ; case 11
        .byte   (.L2 - .Lrtx4) / 4     ; case 12
        .byte   (.L2 - .Lrtx4) / 4     ; case 13
        .byte   (.L2 - .Lrtx4) / 4     ; case 14
        .byte   (.L2 - .Lrtx4) / 4     ; case 15
        .byte   (.L2 - .Lrtx4) / 4     ; case 16
        .byte   (.L2 - .Lrtx4) / 4     ; case 17
        .byte   (.L2 - .Lrtx4) / 4     ; case 18
        .byte   (.L2 - .Lrtx4) / 4     ; case 19
        .byte   (.L6 - .Lrtx4) / 4     ; case 20
        .byte   (.L6 - .Lrtx4) / 4     ; case 21
        .byte   (.L7 - .Lrtx4) / 4     ; case 22
        .text
; tout ce qui se trouve après la déclaration ".text" est alloué dans le
    segment de code (text):
.L7:
; afficher "22"
        adrp    x0, .LC3
        add     x0, x0, :lo12:.LC3
        b       puts
.L6:
; afficher "8, 9, 21"
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        b       puts
.L5:
; afficher "3, 4, 5"
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
        b       puts
.L3:
; afficher "1, 2, 7, 10"
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        b       puts
.LC0:
        .string "1, 2, 7, 10"
.LC1:
        .string "3, 4, 5"
.LC2:
```

```
        .string "8, 9, 21"
.LC3:
        .string "22"
.LC4:
        .string "default"
```

Compilons cet exemple en un fichier objet et ouvrons-le dans IDA. Voici la table des sauts:

Listing 1.165 : jumptable in IDA

```
.rodata:0000000000000064           AREA .rodata, DATA, READONLY
.rodata:0000000000000064           ; ORG 0x64
.rodata:0000000000000064 $d        DCB    9    ; case 1
.rodata:0000000000000065           DCB    9    ; case 2
.rodata:0000000000000066           DCB    6    ; case 3
.rodata:0000000000000067           DCB    6    ; case 4
.rodata:0000000000000068           DCB    6    ; case 5
.rodata:0000000000000069           DCB    6    ; case 6
.rodata:000000000000006A           DCB    9    ; case 7
.rodata:000000000000006B           DCB    3    ; case 8
.rodata:000000000000006C           DCB    3    ; case 9
.rodata:000000000000006D           DCB    9    ; case 10
.rodata:000000000000006E           DCB 0xF7    ; case 11
.rodata:000000000000006F           DCB 0xF7    ; case 12
.rodata:0000000000000070           DCB 0xF7    ; case 13
.rodata:0000000000000071           DCB 0xF7    ; case 14
.rodata:0000000000000072           DCB 0xF7    ; case 15
.rodata:0000000000000073           DCB 0xF7    ; case 16
.rodata:0000000000000074           DCB 0xF7    ; case 17
.rodata:0000000000000075           DCB 0xF7    ; case 18
.rodata:0000000000000076           DCB 0xF7    ; case 19
.rodata:0000000000000077           DCB    3    ; case 20
.rodata:0000000000000078           DCB    3    ; case 21
.rodata:0000000000000079           DCB    0    ; case 22
.rodata:000000000000007B ; .rodata ends
```

Donc dans le cas de 1, 9 est multiplié par 4 et ajouté à l'adresse du label `Lrtx4`.

Dans le cas de 22, 0 est multiplié par 4, ce qui donne 0.

Juste après le label `Lrtx4` se trouve le label L7, où se trouve le code qui affiche «22 ».

Il n'y a pas de table des sauts dans le segment de code, elle est allouée dans la section .rodata (il n'y a pas de raison de l'allouer dans le segment de code).

Il y a aussi des octets négatifs (0xF7), ils sont utilisés pour sauter en arrière dans le code qui affiche la chaîne «default » (en `.L2`).

1.21.4 Fall-through

Un autre usage très répandu de l'opérateur `switch()` est ce qu'on appelle un «fall-through» (passer à travers). Voici un exemple simple[99] :

```
bool is_whitespace(char c) {
	switch (c) {
		case ' ': // fallthrough
		case '\t': // fallthrough
		case '\r': // fallthrough
		case '\n':
			return true;
		default: // not whitespace
			return false;
	}
}
```

Légèrement plus difficile, tiré du noyau Linux[100] :

```
char nco1, nco2;

void f(int if_freq_khz)
{

	switch (if_freq_khz) {
		default:
			printf("IF=%d KHz is not supportted, 3250 assumed\n", if_freq_khz);
			/* fallthrough */
		case 3250: /* 3.25Mhz */
			nco1 = 0x34;
			nco2 = 0x00;
			break;
		case 3500: /* 3.50Mhz */
			nco1 = 0x38;
			nco2 = 0x00;
			break;
		case 4000: /* 4.00Mhz */
			nco1 = 0x40;
			nco2 = 0x00;
			break;
		case 5000: /* 5.00Mhz */
			nco1 = 0x50;
			nco2 = 0x00;
			break;
		case 5380: /* 5.38Mhz */
			nco1 = 0x56;
			nco2 = 0x14;
			break;
	}
```

[99] Copié/collé depuis https://github.com/azonalon/prgraas/blob/master/prog1lib/lecture_examples/is_whitespace.c

[100] Copié/collé depuis https://github.com/torvalds/linux/blob/master/drivers/media/dvb-frontends/lgdt3306a.c

```
};
```

Listing 1.166 : GCC 5.4.0 x86 avec optimisation

```
.LC0:
        .string "IF=%d KHz is not supportted, 3250 assumed\n"
f:
        sub     esp, 12
        mov     eax, DWORD PTR [esp+16]
        cmp     eax, 4000
        je      .L3
        jg      .L4
        cmp     eax, 3250
        je      .L5
        cmp     eax, 3500
        jne     .L2
        mov     BYTE PTR nco1, 56
        mov     BYTE PTR nco2, 0
        add     esp, 12
        ret
.L4:
        cmp     eax, 5000
        je      .L7
        cmp     eax, 5380
        jne     .L2
        mov     BYTE PTR nco1, 86
        mov     BYTE PTR nco2, 20
        add     esp, 12
        ret
.L2:
        sub     esp, 8
        push    eax
        push    OFFSET FLAT:.LC0
        call    printf
        add     esp, 16
.L5:
        mov     BYTE PTR nco1, 52
        mov     BYTE PTR nco2, 0
        add     esp, 12
        ret
.L3:
        mov     BYTE PTR nco1, 64
        mov     BYTE PTR nco2, 0
        add     esp, 12
        ret
.L7:
        mov     BYTE PTR nco1, 80
        mov     BYTE PTR nco2, 0
        add     esp, 12
        ret
```

Nous atteignons le label `.L5` si la fonction a reçue le nombre 3250 en entrée. Mais nous pouvons atteindre ce label d'une autre façon: nous voyons qu'il n'y a pas de

saut entre l'appel à `printf()` et le label `.L5`.

Nous comprenons maintenant pourquoi la déclaration *switch()* est parfois une source de bug: un *break* oublié va transformer notre déclaration *switch()* en un *fallthrough*, et plusieurs blocs seront exécutés au lieu d'un seul.

1.21.5 Exercices

Exercice#1

Il est possible de modifier l'exemple en C de 1.21.2 on page 220 de telle sorte que le compilateur produise un code plus concis, mais qui fonctionne toujours pareil.

1.22 Boucles

1.22.1 Exemple simple

x86

Il y a une instruction LOOP spéciale en x86 qui teste le contenu du registre ECX et si il est différent de 0, le décrémente et continue l'exécution au label de l'opérande LOOP. Probablement que cette instruction n'est pas très pratique, et il n'y a aucun compilateur moderne qui la génère automatiquement. Donc, si vous la rencontrez dans du code, il est probable qu'il s'agisse de code assembleur écrit manuellement.

En C/C++ les boucles sont en général construites avec une déclaration `for()`, `while()` ou `do/while()`.

Commençons avec `for()`.

Cette déclaration définit l'initialisation de la boucle (met le compteur à sa valeur initiale), la condition de boucle (est-ce que le compteur est plus grand qu'une limite?), qu'est-ce qui est fait à chaque itération (incrémenter/décrémenter) et bien sûr le corps de la boucle.

```
for (initialisation; condition; à chaque itération)
{
    corps_de_la_boucle;
}
```

Le code généré consiste également en quatre parties.

Commençons avec un exemple simple:

```
#include <stdio.h>

void printing_function(int i)
{
        printf ("f(%d)\n", i);
};

int main()
{
```

```
        int i;

        for (i=2; i<10; i++)
                printing_function(i);

        return 0;
};
```

Résultat (MSVC 2010) :

Listing 1.167 : MSVC 2010

```
_i$ = -4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR _i$[ebp], 2   ; initialiser la boucle
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp] ; ici se trouve ce que nous faisons après
    chaque itération:
    add    eax, 1                  ; ajouter 1 à la valeur de (i)
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 10  ; cette condition est testée avant chaque
    itération
    jge    SHORT $LN1@main         ; si (i) est supérieur ou égal à 10, la
    boucle se termine
    mov    ecx, DWORD PTR _i$[ebp] ; corps de la boucle: appel de
    printing_function(i)
    push   ecx
    call   _printing_function
    add    esp, 4
    jmp    SHORT $LN2@main         ; saut au début de la boucle
$LN1@main:                         ; fin de la boucle
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Comme nous le voyons, rien de spécial.

GCC 4.4.1 génère presque le même code, avec une différence subtile:

Listing 1.168 : GCC 4.4.1

```
main            proc near

var_20          = dword ptr -20h
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
```

```
                sub     esp, 20h
                mov     [esp+20h+var_4], 2  ; initialiser (i)
                jmp     short loc_8048476

loc_8048465:
                mov     eax, [esp+20h+var_4]
                mov     [esp+20h+var_20], eax
                call    printing_function
                add     [esp+20h+var_4], 1  ; incrémenter (i)

loc_8048476:
                cmp     [esp+20h+var_4], 9
                jle     short loc_8048465   ; si i<=9, continuer la boucle
                mov     eax, 0
                leave
                retn
main            endp
```

Maintenant, regardons ce que nous obtenons avec l'optimisation (/Ox) :

Listing 1.169 : avec optimisation MSVC

```
_main    PROC
    push   esi
    mov    esi, 2
$LL3@main:
    push   esi
    call   _printing_function
    inc    esi
    add    esp, 4
    cmp    esi, 10     ; 0000000aH
    jl     SHORT $LL3@main
    xor    eax, eax
    pop    esi
    ret    0
_main    ENDP
```

Ce qui se passe alors, c'est que l'espace pour la variable i n'est plus alloué sur la pile locale, mais utilise un registre individuel pour cela, ESI. Ceci est possible pour ce genre de petites fonctions, où il n'y a pas beaucoup de variables locales.

Il est très important que la fonction f() ne modifie pas la valeur de ESI. Notre compilateur en est sûr ici. Et si le compilateur décide d'utiliser le registre ESI aussi dans la fonction f(), sa valeur devra être sauvegardée lors du prologue de la fonction et restaurée lors de son épilogue, presque comme dans notre listing: notez les PUSH ESI/POP ESI au début et à la fin de la fonction.

Essayons GCC 4.4.1 avec l'optimisation la plus performante (option -O3) :

Listing 1.170 : GCC 4.4.1 avec optimisation

```
main            proc near

var_10          = dword ptr -10h
```

```
                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     [esp+10h+var_10], 2
                call    printing_function
                mov     [esp+10h+var_10], 3
                call    printing_function
                mov     [esp+10h+var_10], 4
                call    printing_function
                mov     [esp+10h+var_10], 5
                call    printing_function
                mov     [esp+10h+var_10], 6
                call    printing_function
                mov     [esp+10h+var_10], 7
                call    printing_function
                mov     [esp+10h+var_10], 8
                call    printing_function
                mov     [esp+10h+var_10], 9
                call    printing_function
                xor     eax, eax
                leave
                retn
main            endp
```

Hé, GCC a juste déroulé notre boucle.

Le déroulement de boucle est un avantage lorsqu'il n'y a pas beaucoup d'itérations et que nous pouvons économiser du temps d'exécution en supprimant les instructions de gestion de la boucle. D'un autre côté, le code est étonnement plus gros.

Dérouler des grandes boucles n'est pas recommandé de nos jours, car les grosses fonctions ont une plus grande empreinte sur le cache[101].

Ok, augmentons la valeur maximale de la variable i à 100 et essayons à nouveau. GCC donne:

Listing 1.171 : GCC

```
                public main
main            proc near

var_20          = dword ptr -20h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                push    ebx
                mov     ebx, 2    ; i=2
                sub     esp, 1Ch
```

[101]Un très bon article à ce sujet: [Ulrich Drepper, *What Every Programmer Should Know About Memory*, (2007)][102]. D'autres recommandations sur l'expansion des boucles d'Intel sont ici: [*Intel® 64 and IA-32 Architectures Optimization Reference Manual*, (2014)3.4.1.7].

```
; aligner le label loc_80484D0 (début du corps de la boucle) sur une limite
    de 16-octet:
                nop

loc_80484D0:
; passer (i) comme premier argument à printing_function():
                mov     [esp+20h+var_20], ebx
                add     ebx, 1    ; i++
                call    printing_function
                cmp     ebx, 64h  ; i==100?
                jnz     short loc_80484D0 ; si non, continuer
                add     esp, 1Ch
                xor     eax, eax  ; renvoyer 0
                pop     ebx
                mov     esp, ebp
                pop     ebp
                retn
main            endp
```

C'est assez similaire à ce que MSVC 2010 génère avec l'optimisation (/Ox), avec l'exception que le registre EBX est utilisé pour la variable i.

GCC est sûr que ce registre ne sera pas modifié à l'intérieur de la fonction `f()`, et si il l'était, il serait sauvé dans le prologue de la fonction et restauré dans l'épilogue, tout comme dans la fonction `main()`.

x86: OllyDbg

Compilons notre exemple dans MSVC 2010 avec les options `/Ox` et `/Ob0`, puis chargeons le dans OllyDbg.

Il semble qu'OllyDbg soit capable de détecter des boucles simples et les affiche entre parenthèses, par commodité.

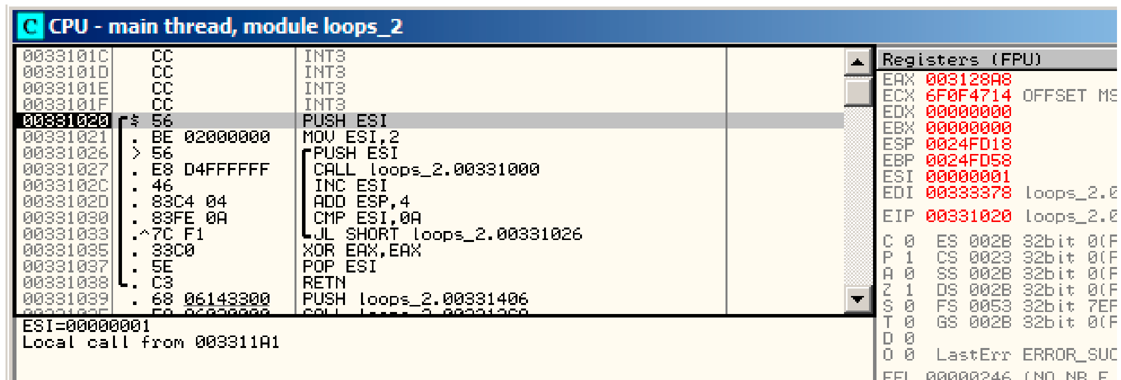


Fig. 1.54: OllyDbg : début de `main()`

En traçant (F8 — enjamber) nous voyons ESI s'incrémenter. Ici, par exemple, $ESI = i = 6$:

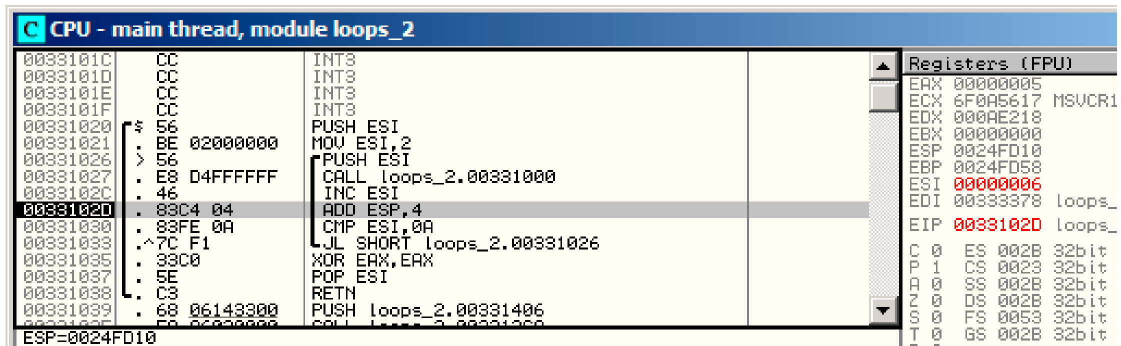


Fig. 1.55: OllyDbg : le corps de la boucle vient de s'exécuter avec $i = 6$

9 est la dernière valeur de la boucle. C'est pourquoi JL ne s'exécute pas après l'incrémentation, et que le fonction se termine.

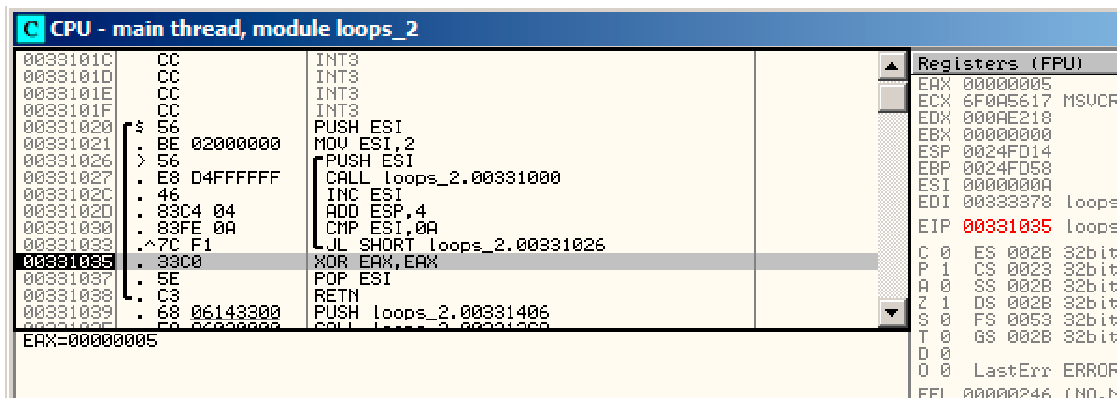


Fig. 1.56: OllyDbg : $ESI = 10$, fin de la boucle

x86: tracer

Comme nous venons de le voir, il n'est pas très commode de tracer manuellement dans le débogueur. C'est pourquoi nous allons essayer tracer.

Nous ouvrons dans IDA l'exemple compilé, trouvons l'adresse de l'instruction PUSH ESI (qui passe le seul argument à `f()`), qui est `0x401026` dans ce cas et nous lançons le tracer :

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

BPX met juste un point d'arrêt à l'adresse et tracer va alors afficher l'état des registres.

Voici ce que l'on voit dans `tracer.log` :

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x00a328c8 EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x00000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
```

```
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000009 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
PID=12884|Process loops_2.exe exited. ExitCode=0 (0x0)
```

Nous voyons comment la valeur du registre ESI change de 2 à 9.

Encore plus que ça, tracer peut collecter les valeurs des registres pour toutes les adresses dans la fonction. C'est appelé *trace* ici. Chaque instruction est tracée, toutes les valeurs intéressantes des registres sont enregistrées.

Ensuite, un script IDA .idc est généré, qui ajoute des commentaires. Donc, dans IDA, nous avons appris que l'adresse de la fonction `main()` est `0x00401020` et nous lançons:

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

BPF signifie mettre un point d'arrêt sur la fonction.

Comme résultat, nous obtenons les scripts `loops_2.exe.idc` et `loops_2.exe_clear.idc`.

Nous chargeons loops_2.exe.idc dans IDA et voyons:

```
.text:00401020
.text:00401020 ; =============== S U B R O U T I N E =======================================
.text:00401020
.text:00401020
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main           proc near               ; CODE XREF: ___tmainCRTStartup+11D↓p
.text:00401020
.text:00401020 argc            = dword ptr  4
.text:00401020 argv            = dword ptr  8
.text:00401020 envp            = dword ptr  0Ch
.text:00401020
.text:00401020                 push    esi             ; ESI=1
.text:00401021                 mov     esi, 2
.text:00401026
.text:00401026 loc_401026:                             ; CODE XREF: _main+13↓j
.text:00401026                 push    esi             ; ESI=2..9
.text:00401027                 call    sub_401000      ; tracing nested maximum level (1) reached,
.text:0040102C                 inc     esi             ; ESI=2..9
.text:0040102D                 add     esp, 4          ; ESP=0x38fcbc
.text:00401030                 cmp     esi, 0Ah        ; ESI=3..0xa
.text:00401033                 jl      short loc_401026 ; SF=false,true OF=false
.text:00401035                 xor     eax, eax
.text:00401037                 pop     esi
.text:00401038                 retn                    ; EAX=0
.text:00401038 _main           endp
```

Fig. 1.57: IDA avec le script .idc chargé

Nous voyons que ESI varie de 2 à 9 au début du corps de boucle, mais de 3 à 0xA (10) après l'incrément. Nous voyons aussi que main() se termine avec 0 dans EAX.

tracer génère également loops_2.exe.txt, qui contient des informations sur le nombre de fois qu'une instruction a été exécutée et les valeurs du registre:

Listing 1.172 : loops_2.exe.txt

```
0x401020 (.text+0x20), e=       1 [PUSH ESI] ESI=1
0x401021 (.text+0x21), e=       1 [MOV ESI, 2]
0x401026 (.text+0x26), e=       8 [PUSH ESI] ESI=2..9
0x401027 (.text+0x27), e=       8 [CALL 8D1000h] tracing nested maximum ↙
    ↘ level (1) reached, skipping this CALL 8D1000h=0x8d1000
0x40102c (.text+0x2c), e=       8 [INC ESI] ESI=2..9
0x40102d (.text+0x2d), e=       8 [ADD ESP, 4] ESP=0x38fcbc
0x401030 (.text+0x30), e=       8 [CMP ESI, 0Ah] ESI=3..0xa
0x401033 (.text+0x33), e=       8 [JL 8D1026h] SF=false,true OF=false
0x401035 (.text+0x35), e=       1 [XOR EAX, EAX]
0x401037 (.text+0x37), e=       1 [POP ESI]
0x401038 (.text+0x38), e=       1 [RETN] EAX=0
```

Nous pouvons utiliser grep ici.

ARM

sans optimisation Keil 6/2013 (Mode ARM)

```
main
        STMFD   SP!, {R4,LR}
        MOV     R4, #2
        B       loc_368
loc_35C  ; CODE XREF: main+1C
        MOV     R0, R4
        BL      printing_function
        ADD     R4, R4, #1

loc_368  ; CODE XREF: main+8
        CMP     R4, #0xA
        BLT     loc_35C
        MOV     R0, #0
        LDMFD   SP!, {R4,PC}
```

Le compteur de boucle i est stocké dans le registre R4. L'instruction `MOV R4, #2` initialise i. Les instructions `MOV R0, R4` et `BL printing_function` composent le corps de la boucle, la première instruction préparant l'argument pour la fonction `f()` et la seconde l'appelant. L'instruction `ADD R4, R4, #1` ajoute 1 à la variable i à chaque itération. `CMP R4, #0xA` compare i avec `0xA` (10). L'instruction suivante, BLT (*Branch Less Than*) saute si i est inférieur à 10. Autrement, 0 est écrit dans `R0` (puisque notre fonction renvoie 0) et l'exécution de la fonction se termine.

avec optimisation Keil 6/2013 (Mode Thumb)

```
_main
                PUSH    {R4,LR}
                MOVS    R4, #2

loc_132                                 ; CODE XREF: _main+E
                MOVS    R0, R4
                BL      printing_function
                ADDS    R4, R4, #1
                CMP     R4, #0xA
                BLT     loc_132
                MOVS    R0, #0
                POP     {R4,PC}
```

Pratiquement la même chose.

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
_main
       PUSH            {R4,R7,LR}
       MOVW            R4, #0x1124 ; "%d\n"
       MOVS            R1, #2
       MOVT.W          R4, #0
       ADD             R7, SP, #4
       ADD             R4, PC
```

```
        MOV             R0, R4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #3
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #5
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #6
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #7
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #8
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #9
        BLX             _printf
        MOVS            R0, #0
        POP             {R4,R7,PC}
```

En fait, il y avait ceci dans ma fonction `f()` :

```
void printing_function(int i)
{
    printf ("%d\n", i);
};
```

Donc, non seulement LLVM *déroule* la boucle, mais aussi *inline* ma fonction très simple et insère son corps 8 fois au lieu de l'appeler.

Ceci est possible lorsque la fonction est très simple (comme la mienne) et lorsqu'elle n'est pas trop appelée (comme ici).

ARM64: GCC 4.9.1 avec optimisation

Listing 1.173 : GCC 4.9.1 avec optimisation

```
printing_function:
; préparer le second argument de printf():
        mov     w1, w0
; charger l'adresse de la chaîne "f(%d)\n"
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; seulement sauter ici au lieu de sauter avec lien et retour:
        b       printf
main:
```

```
; sauver FP et LR dans la pile locale:
        stp     x29, x30, [sp, -32]!
; préparer une structure de pile:
        add     x29, sp, 0
; sauver le contenu du registre X19 dans la pile locale:
        str     x19, [sp,16]
; nous allons utiliser le registre W19 comme compteur.
; lui assigner une valeur initiale de 2:
        mov     w19, 2
.L3:
; préparer le premier argument de printing_function():
        mov     w0, w19
; incrémenter le registre compteur.
        add     w19, w19, 1
; ici W0 contient toujours la valeur du compteur avant incrémentation.
        bl      printing_function
; est-ce terminé?
        cmp     w19, 10
; non, sauter au début du corps de boucle:
        bne     .L3
; renvoyer 0
        mov     w0, 0
; restaurer le contenu du registre X19:
        ldr     x19, [sp,16]
; restaurer les valeurs de FP et LR:
        ldp     x29, x30, [sp], 32
        ret
.LC0:
        .string "f(%d)\n"
```

ARM64: GCC 4.9.1 sans optimisation

Listing 1.174 : GCC 4.9.1 -fno-inline sans optimisation

```
.LC0:
        .string "f(%d)\n"
printing_function:
; sauver FP et LR dans la pile locale:
        stp     x29, x30, [sp, -32]!
; préparer la pile locale:
        add     x29, sp, 0
; sauver le contenu du registre W0:
        str     w0, [x29,28]
; charger l'adresse de la chaîne "f(%d)\n"
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; recharger la valeur entrée depuis le pile locale dans le registre W0:
        ldr     w1, [x29,28]
; appeler printf()
        bl      printf
; restaurer les valeurs de FP et LR:
        ldp     x29, x30, [sp], 32
```

```
        ret
main:
; sauvegarder FP et LR sur la pile locale:
        stp     x29, x30, [sp, -32]!
; préparer la structure de pile:
        add     x29, sp, 0
; initialiser le compteur
        mov     w0, 2
; le stocker dans l'espace alloué pour lui dans la pile locale:
        str     w0, [x29,28]
; passer le corps de la boucle et sauter aux instructions de vérification de
    la condition de boucle:
        b       .L3
.L4:
; charger la valeur du compteur dans W0.
; ce sera le premier argument de printing_function():
        ldr     w0, [x29,28]
; appeler printing_function():
        bl      printing_function
; incrémenter la valeur du compteur:
        ldr     w0, [x29,28]
        add     w0, w0, 1
        str     w0, [x29,28]
.L3:
; tester condition de boucle.
; charger la valeur du compteur:
        ldr     w0, [x29,28]
; est-ce 9?
        cmp     w0, 9
; inférieur ou égal? alors sauter au début du corps de boucle:
; autrement, ne rien faire.
        ble     .L4
; renvoyer 0
        mov     w0, 0
; restaurer les valeurs de FP et LR:
        ldp     x29, x30, [sp], 32
        ret
```

MIPS

Listing 1.175 : GCC 4.4.5 sans optimisation (IDA)

```
main:

; IDA ne connaît pas le nom des variables dans la pile locale
; Nous pouvons leurs en donner un manuellement:
i               = -0x10
saved_FP        = -8
saved_RA        = -4

; prologue de la fonction:
                addiu   $sp, -0x28
                sw      $ra, 0x28+saved_RA($sp)
```

```
                sw      $fp, 0x28+saved_FP($sp)
                move    $fp, $sp
; initialiser le compteur à 2 et stocker cette valeur dans la pile locale
                li      $v0, 2
                sw      $v0, 0x28+i($fp)
; pseudo-instruction. "BEQ $ZERO, $ZERO, loc_9C" c'est en fait:
                b       loc_9C
                or      $at, $zero ; slot de délai de branchement, NOP

loc_80:                                  # CODE XREF: main+48
; charger la valeur du compteur depuis la pile locale et appeler
    printing_function():
                lw      $a0, 0x28+i($fp)
                jal     printing_function
                or      $at, $zero ; slot de délai de branchement, NOP
; charger le compteur, l'incrémenter, et le stocker de nouveau:
                lw      $v0, 0x28+i($fp)
                or      $at, $zero ; NOP
                addiu   $v0, 1
                sw      $v0, 0x28+i($fp)

loc_9C:                                  # CODE XREF: main+18
; tester le compteur, est-ce 10?
                lw      $v0, 0x28+i($fp)
                or      $at, $zero ; NOP
                slti    $v0, 0xA
; si il est inférieur à 10, sauter en loc_80 (début du corps de la boucle):
                bnez    $v0, loc_80
                or      $at, $zero ; slot de délai de branchement, NOP
; fin, renvoyer 0:
                move    $v0, $zero
; épilogue de la fonction:
                move    $sp, $fp
                lw      $ra, 0x28+saved_RA($sp)
                lw      $fp, 0x28+saved_FP($sp)
                addiu   $sp, 0x28
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

L'instruction qui est nouvelle pour nous est B. C'est la pseudo instruction (BEQ).

Encore une chose

Dans le code généré, nous pouvons voir: après avoir initialisé i, le corps de la boucle n'est pas exécuté, car la condition sur i est d'abord vérifiée, et c'est seulement après cela que le corps de la boucle peut être exécuté. Et cela est correct.

Ceci car si la condition de boucle n'est pas remplie au début, le corps de la boucle ne doit pas être exécuté. Ceci est possible dans le cas suivant:

```
for (i=0; i<nombre_total_d_element_à_traiter; i++)
    corps_de_la_boucle;
```

Si *nombre_total_d_element_à_traiter* est 0, le corps de la boucle ne sera pas exécuté du tout.

C'est pourquoi la condition est testée avant l'exécution.

Toutefois, un compilateur qui optimise pourrait échanger le corps de la boucle et la condition, si il est certain que la situation que nous venons de décrire n'est pas possible (comme dans le cas de notre exemple simple, et en utilisant des compilateurs comme Keil, Xcode (LLVM) et MSVC avec le flag d'optimisation.

1.22.2 Routine de copie de blocs de mémoire

Les routines réelles de copie de mémoire copient 4 ou 8 octets à chaque itération, utilisent SIMD[103], la vectorisation, etc. Mais dans un but didactique, cet exemple est le plus simple possible.

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
        size_t i;
        for (i=0; i<cnt; i++)
                dst[i]=src[i];
};
```

Implémentation simple

Listing 1.176 : GCC 4.9 x64 optimisé pour la taille (-Os)

```
my_memcpy:
; RDI = adresse de destination
; RSI = adresse source
; RDX = taille de bloc

; initialiser le compteur (i) à 0
        xor     eax, eax
.L2:
; tous les octets sont-ils copiés? alors sortir:
        cmp     rax, rdx
        je      .L5
; charger l'octet en RSI+i:
        mov     cl, BYTE PTR [rsi+rax]
; stocker l'octet en RDI+i:
        mov     BYTE PTR [rdi+rax], cl
        inc     rax ; i++
        jmp     .L2
.L5:
        ret
```

[103] Single Instruction, Multiple Data

Listing 1.177 : GCC 4.9 ARM64 optimisé pour la taille (-Os)

```
my_memcpy:
; X0 = adresse de destination
; X1 = adresse source
; X2 = taille de bloc

; initialiser le compteur (i) à 0
        mov     x3, 0
.L2:
; tous les octets sont-ils copiés? alors sortir:
        cmp     x3, x2
        beq     .L5
; charger l'octet en X1+i:
        ldrb    w4, [x1,x3]
; stocker l'octet en X0+i:
        strb    w4, [x0,x3]
        add     x3, x3, 1 ; i++
        b       .L2
.L5:
        ret
```

Listing 1.178 : avec optimisation Keil 6/2013 (Mode Thumb)

```
my_memcpy PROC
; R0 = adresse de destination
; R1 = adresse source
; R2 = taille de bloc

        PUSH     {r4,lr}
; initialiser le compteur (i) à 0
        MOVS     r3,#0
; la condition est testée à la fin de la fonction, donc y sauter:
        B        |L0.12|
|L0.6|
; charger l'octet en R1+i:
        LDRB     r4,[r1,r3]
; stocker l'octet en R0+i:
        STRB     r4,[r0,r3]
; i++
        ADDS     r3,r3,#1
|L0.12|
; i<taille?
        CMP      r3,r2
; sauter au début de la boucle si c'est le cas:
        BCC      |L0.6|
        POP      {r4,pc}
        ENDP
```

ARM en mode ARM

Keil en mode ARM tire pleinement avantage des suffixes conditionnels:

Listing 1.179 : avec optimisation Keil 6/2013 (Mode ARM)

```
my_memcpy PROC
; R0 = adresse de destination
; R1 = adresse source
; R2 = taille de bloc

; initialiser le compteur (i) à 0
        MOV      r3,#0
|L0.4|
; tous les octets sont-ils copiés?
        CMP      r3,r2
; le bloc suivant est exécuté seulement si la condition less than est
    remplie,
; i.e., if R2<R3 ou i<taille.
; charger l'octet en R1+i:
        LDRBCC   r12,[r1,r3]
; stocker l'octet en R0+i:
        STRBCC   r12,[r0,r3]
; i++
        ADDCC    r3,r3,#1
; la dernière instruction du bloc conditionnel.
; sauter au début de la boucle si i<taille
; ne rien faire autrement (i.e., si i>=taille)
        BCC      |L0.4|
; retourner
        BX       lr
        ENDP
```

C'est pourquoi il y a seulement une instruction de branchement au lieu de 2.

MIPS

Listing 1.180 : GCC 4.4.5 optimisé pour la taille (-Os) (IDA)

```
my_memcpy:
; sauter à la partie test de la boucle:
                b       loc_14
; initialiser le compteur (i) à 0
; il se trouvera toujours dans $v0:
                move    $v0, $zero ; slot de délai de branchement

loc_8:                                  # CODE XREF: my_memcpy+1C
; charger l'octet non-signé à l'adresse $t0 dans $v1:
                lbu     $v1, 0($t0)
; incrémenter le compteur (i):
                addiu   $v0, 1
; stocker l'octet en $a3
                sb      $v1, 0($a3)

loc_14:                                 # CODE XREF: my_memcpy
; tester si le compteur (i) dans $v0 est toujours inférieur au 3ème argument
    de la fonction ("cnt" dans $a2):
                sltu    $v1, $v0, $a2
```

```
; former l'adresse de l'octet dans le bloc source:
                addu    $t0, $a1, $v0
; $t0 = $a1+$v0 = src+i
; sauter au corps de la boucle si le compteur est toujours inférieur à "cnt":
                bnez    $v1, loc_8
; former l'adresse de l'octet dans le bloc de destination ($a3 = $a0+$v0 =
    dst+i):
                addu    $a3, $a0, $v0 ; slot de délai de branchement
; terminer si BNEZ n'a pas exécuté de saut:
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

Nous avons ici deux nouvelles instructions: LBU («Load Byte Unsigned» charger un octet non signé) et SB («Store Byte» stocker un octet).

Tout comme en ARM, tous les registres MIPS ont une taille de 32-bit, il n'y en a pas d'un octet de large comme en x86.

Donc, lorsque l'on travaille avec des octets seuls, nous devons utiliser un registre de 32-bit pour chacun d'entre eux.

LBU charge un octet et met les autres bits à zéro («Unsigned»).

En revanche, l'instruction LB («Load Byte») étend le signe de l'octet chargé sur 32-bit.

SB écrit simplement un octet depuis les 8 bits de poids faible d'un registre dans la mémoire.

Vectorisation

GCC avec optimisation peut faire beaucoup mieux avec cet exemple: 1.36.1 on page 534.

1.22.3 Vérification de condition

Il est important de garder à l'esprit que dans une boucle *for()*, la condition est vérifiée préalablement à l'itération du corps de la boucle et non pas après. Cela étant il est souvent plus pratique pour le compilateur de placer les instructions qui effectuent le test après le corps de la boucle. Il arrive aussi qu'il rajoute des vérifications au début du corps de la boucle.

Par exemple:

```
#include <stdio.h>

void f(int start, int finish)
{
        for (; start<finish; start++)
                printf ("%d\n", start);
};
```

GCC 5.4.0 x64 en mode optimisé:

```
f:
; check condition (1):
        cmp     edi, esi
        jge     .L9
        push    rbp
        push    rbx
        mov     ebp, esi
        mov     ebx, edi
        sub     rsp, 8
.L5:
        mov     edx, ebx
        xor     eax, eax
        mov     esi, OFFSET FLAT:.LC0 ; "%d\n"
        mov     edi, 1
        add     ebx, 1
        call    __printf_chk
; check condition (2):
        cmp     ebp, ebx
        jne     .L5
        add     rsp, 8
        pop     rbx
        pop     rbp
.L9:
        rep ret
```

Nous constatons la présence de deux vérifications.

Le code décompilé produit par Hex-Rays (dans sa version 2.2.0) est celui-ci:

```
void __cdecl f(unsigned int start, unsigned int finish)
{
  unsigned int v2; // ebx@2
  __int64 v3; // rdx@3

  if ( (signed int)start < (signed int)finish )
  {
    v2 = start;
    do
    {
      v3 = v2++;
      _printf_chk(1LL, "%d\n", v3);
    }
    while ( finish != v2 );
  }
}
```

Dans le cas présent, il ne fait aucun doute que la structure *do/while()* peut être remplacée par une construction *for()*, et que le premier contrôle peut être supprimé.

1.22.4 Conclusion

Squelette grossier d'une boucle de 2 à 9 inclus:

Listing 1.181 : x86

```
    mov [counter], 2 ; initialisation
    jmp check
body:
    ; corps de la boucle
    ; faire quelque chose ici
    ; utiliser la variable compteur dans la pile locale
    add [counter], 1 ; incrémenter
check:
    cmp [counter], 9
    jle body
```

L'opération d'incrémentation peut être représentée par 3 instructions dans du code non optimisé:

Listing 1.182 : x86

```
    MOV [counter], 2 ; initialisation
    JMP check
body:
    ; corps de la boucle
    ; faire quelque chose ici
    ; utiliser la variable compteur dans la pile locale
    MOV REG, [counter] ; incrémenter
    INC REG
    MOV [counter], REG
check:
    CMP [counter], 9
    JLE body
```

Si le corps de la boucle est court, un registre entier peut être dédié à la variable compteur:

Listing 1.183 : x86

```
    MOV EBX, 2 ; initialisation
    JMP check
body:
    ; corps de la boucle
    ; faire quelque chose ici
    ; utiliser le compteur dans EBX, mais ne pas le modifier!
    INC EBX ; incrémenter
check:
    CMP EBX, 9
    JLE body
```

Certaines parties de la boucle peuvent être générées dans un ordre différent par le compilateur:

Listing 1.184 : x86

```
    MOV [counter], 2 ; initialisation
    JMP label_check
label_increment:
```

```
    ADD [counter], 1 ; incrémenter
label_check:
    CMP [counter], 10
    JGE exit
    ; corps de la boucle
    ; faire quelque cose ici
    ; utiliser la variable compteur dans la pile locale
    JMP label_increment
exit:
```

En général, la condition est testée *avant* le corps de la boucle, mais le compilateur peut la réarranger afin que la condition soit testée *après* le corps de la boucle.

Cela est fait lorsque le compilateur est certain que la condition est toujours *vraie* à la première itération, donc que le corps de la boucle doit être exécuté au moins une fois:

Listing 1.185 : x86

```
    MOV REG, 2 ; initialisation
body:
    ; corps de la boucle
    ; faire quelque chose ici
    ; utiliser le compteur dans REG, mais ne pas le modifier!
    INC REG ; incrémenter
    CMP REG, 10
    JL body
```

En utilisant l'instruction LOOP. Ceci est rare, les compilateurs ne l'utilisent pas. Lorsque vous la voyez, c'est le signe que le morceau de code a été écrit à la main:

Listing 1.186 : x86

```
    ; compter de 10 à 1
    MOV ECX, 10
body:
    ; corps de la boucle
    ; faire quelque chose ici
    ; utiliser le compteur dans ECX, mais ne pas le modifier!
    LOOP body
```

ARM.

Le registre R4 est dédié à la variable compteur dans cet exemple:

Listing 1.187 : ARM

```
    MOV R4, 2 ; initialisation
    B check
body:
    ; corps de la boucle
    ; faire quelque chose ici
    ; utiliser le compteur dans R4, mais ne pas le modifier!
    ADD R4,R4, #1 ; incrémenter
check:
```

```
    CMP R4, #10
    BLT body
```

1.22.5 Exercices

- http://challenges.re/54
- http://challenges.re/55
- http://challenges.re/56
- http://challenges.re/57

1.23 Plus d'information sur les chaînes

1.23.1 strlen()

Parlons encore une fois des boucles. Souvent, la fonction `strlen()` [104] est implémentée en utilisant une déclaration `while()`. Voici comment cela est fait dans les bibliothèques standards de MSVC:

```
int my_strlen (const char * str)
{
        const char *eos = str;

        while( *eos++ ) ;

        return( eos - str - 1 );
}

int main()
{
        // test
        return my_strlen("hello!");
};
```

x86

MSVC sans optimisation

Compilons:

```
_eos$ = -4                      ; size = 4
_str$ = 8                       ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
```

[104] compter les caractères d'une chaîne en langage C

```
    mov     eax, DWORD PTR _str$[ebp]  ; copier le pointeur sur la chaîne
    "str"
    mov     DWORD PTR _eos$[ebp], eax  ; le copier dans la variable locale
    "eos"
$LN2@strlen_:
    mov     ecx, DWORD PTR _eos$[ebp]  ; ECX=eos

    ; prendre un octet 8-bit depuis l'adresse dans ECX et le copier
    ; comme une valeur 32-bit dans EDX avec extension du signe

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp]  ; EAX=eos
    add     eax, 1                     ; incrémenter EAX
    mov     DWORD PTR _eos$[ebp], eax  ; remettre EAX dans "eos"
    test    edx, edx                   ; est-ce que EDX est à zéro?
    je      SHORT $LN1@strlen_         ; oui, alors finir la boucle
    jmp     SHORT $LN2@strlen_         ; continuer la boucle
$LN1@strlen_:

    ; ici nous calculons la différence entre deux pointeurs

    mov    eax, DWORD PTR _eos$[ebp]
    sub    eax, DWORD PTR _str$[ebp]
    sub    eax, 1                      ; soustraire 1 du résultat et sortir
    mov    esp, ebp
    pop    ebp
    ret    0
_strlen_ ENDP
```

Nous avons ici deux nouvelles instructions: MOVSX et TEST.

La première—MOVSX—prend un octet depuis une adresse en mémoire et stocke la valeur dans un registre 32-bit. MOVSX signifie *MOV with Sign-Extend* (déplacement avec extension de signe). MOVSX met le reste des bits, du 8ème au 31ème, à 1 si l'octet source est *négatif* ou à 0 si il est *positif*.

Et voici pourquoi.

Par défaut, le type *char* est signé dans MSVC et GCC. Si nous avons deux valeurs dont l'une d'elle est un *char* et l'autre un *int*, (*int* est signé aussi), et si la première valeur contient -2 (codé en 0xFE) et que nous copions simplement cet octet dans le conteneur *int*, cela fait 0x000000FE, et ceci, pour le type *int* représente 254, mais pas -2. Dans un entier signé, -2 est codé en 0xFFFFFFFE. Donc, si nous devons transférer 0xFE depuis une variable de type *char* vers une de type *int*, nous devons identifier son signe et l'étendre. C'est ce qu'effectue MOVSX.

Il est difficile de dire si le compilateur doit stocker une variable *char* dans EDX, il pourrait simplement utiliser une partie 8-bit du registre (par exemple DL). Apparemment, l'allocateur de registre fonctionne comme ça.

Ensuite nous voyons TEST EDX, EDX. Vous pouvez en lire plus à propos de l'instruction TEST dans la section concernant les champs de bit (1.28 on page 392). Ici cette instruction teste simplement si la valeur dans EDX est égale à 0.

GCC sans optimisation

Essayons GCC 4.4.1:

```
                public strlen
strlen          proc near

eos             = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+eos], eax

loc_80483F0:
                mov     eax, [ebp+eos]
                movzx   eax, byte ptr [eax]
                test    al, al
                setnz   al
                add     [ebp+eos], 1
                test    al, al
                jnz     short loc_80483F0
                mov     edx, [ebp+eos]
                mov     eax, [ebp+arg_0]
                mov     ecx, edx
                sub     ecx, eax
                mov     eax, ecx
                sub     eax, 1
                leave
                retn
strlen          endp
```

Le résultat est presque le même qu'avec MSVC, mais ici nous voyons `MOVZX` au lieu de `MOVSX`. `MOVZX` signifie *MOV with Zero-Extend* (déplacement avec extension à 0). Cette instruction copie une valeur 8-bit ou 16-bit dans un registre 32-bit et met les bits restant à 0. En fait, cette instructions n'est pratique que pour nous permettre de remplacer cette paire d'instructions:
`xor eax, eax / mov al, [...]`.

D'un autre côté, il est évident que le compilateur pourrait produire ce code:
`mov al, byte ptr [eax] / test al, al`—c'est presque le même, toutefois, les bits les plus haut du registre EAX vont contenir des valeurs aléatoires. Mais, admettons que c'est un inconvénient du compilateur—-il ne peut pas produire du code plus compréhensible. À strictement parler, le compilateur n'est pas du tout obligé de générer du code compréhensible par les humains.

La nouvelle instruction suivante est SETNZ. Ici, si AL ne contient pas zéro, `test al, al` met le flag ZF à 0, mais SETNZ, si `ZF==0` (*NZ* signifie *not zero*, non zéro) met AL à 1. En langage naturel, *si AL n'est pas zéro, sauter en loc_80483F0*. Le compilateur génère du code redondant, mais n'oublions pas qu'il n'est pas en mode optimisation.

MSVC avec optimisation

Maintenant, compilons tout cela avec MSVC 2012, avec le flag d'optimisation (/Ox) :

Listing 1.188 : MSVC 2012 avec optimisation/Ob0

```
_str$ = 8                          ; size = 4
_strlen PROC
        mov     edx, DWORD PTR _str$[esp-4] ; EDX -> pointeur sur la chaîne
        mov     eax, edx                    ; déplacer dans EAX
$LL2@strlen:
        mov     cl, BYTE PTR [eax]          ; CL = *EAX
        inc     eax                         ; EAX++
        test    cl, cl                      ; CL==0?
        jne     SHORT $LL2@strlen           ; non, continuer la boucle
        sub     eax, edx                    ; calculer la différence entre
    les pointeurs
        dec     eax                         ; décrémenter EAX
        ret     0
_strlen ENDP
```

C'est plus simple maintenant. Inutile de préciser que le compilateur ne peut utiliser les registres aussi efficacement que dans une petite fonction, avec peu de variables locales.

INC/DEC—sont des instructions de incrémentation/décrémentation, en d'autres mots: ajouter ou soustraire 1 d'une/à une variable.

MSVC avec optimisation + OllyDbg

Nous pouvons essayer cet exemple (optimisé) dans OllyDbg. Voici la première itération:

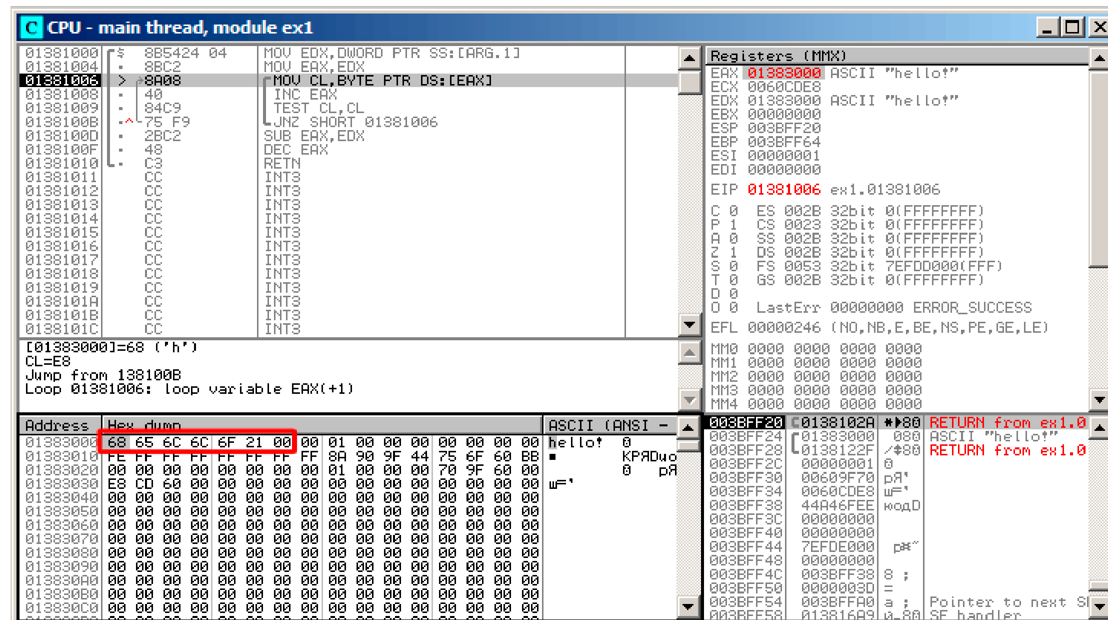


Fig. 1.58: OllyDbg : début de la première itération

Nous voyons qu'OllyDbg a trouvé une boucle et, par facilité, a mis ses instructions entre crochets. En cliquant sur le bouton droit sur EAX, nous pouvons choisir «Follow in Dump» et la fenêtre de la mémoire se déplace jusqu'à la bonne adresse. Ici, nous voyons la chaîne «hello!» en mémoire. Il y a au moins un zéro après cette dernière et ensuite des données aléatoires.

Si OllyDbg voit un registre contenant une adresse valide, qui pointe sur une chaîne, il montre cette chaîne.

Appuyons quelques fois sur F8 (enjamber), pour aller jusqu'au début du corps de la boucle:

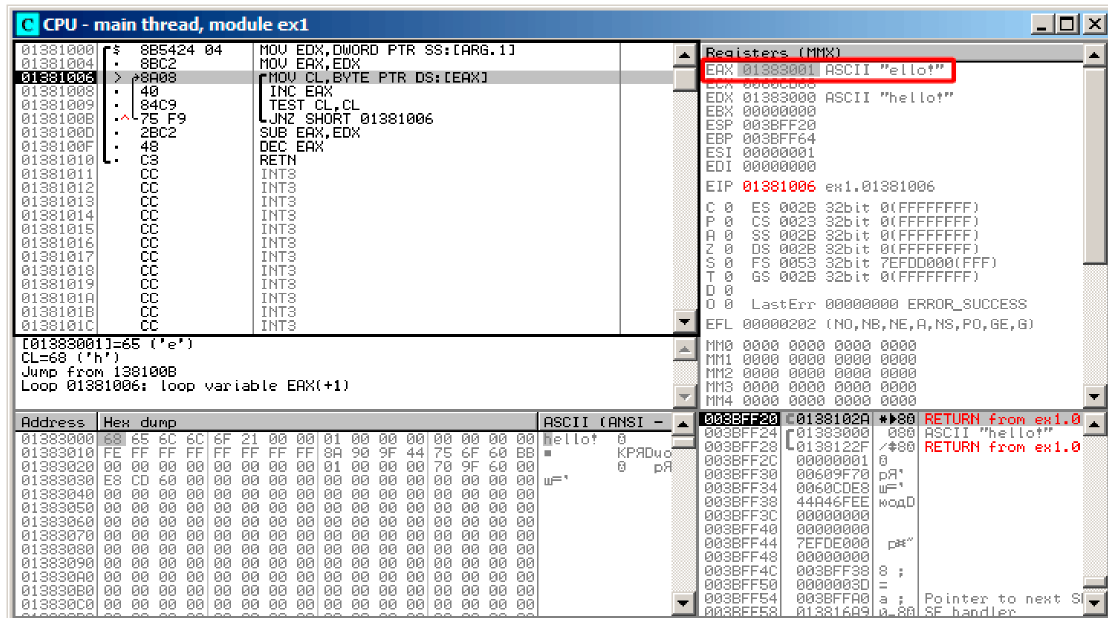


Fig. 1.59: OllyDbg : début de la seconde itération

Nous voyons qu'EAX contient l'adresse du second caractère de la chaîne.

Nous devons appuyons un certain nombre de fois sur F8 afin de sortir de la boucle:

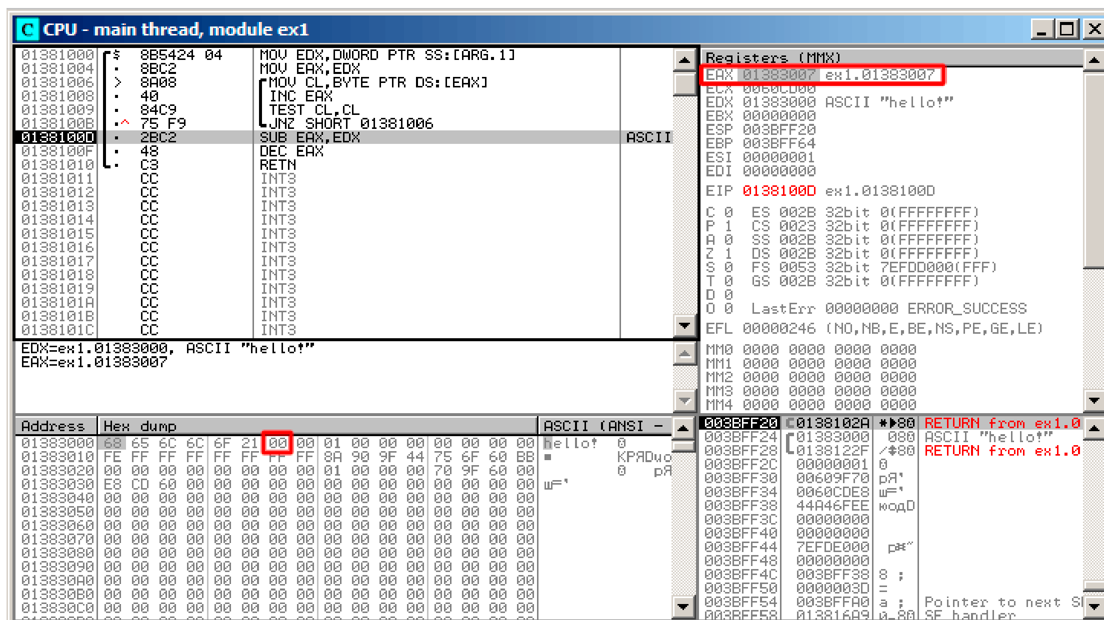


Fig. 1.60: OllyDbg : calcul de la différence entre les pointeurs

Nous voyons qu'EAX contient l'adresse de l'octet à zéro situé juste après la chaîne. Entre temps, EDX n'a pas changé, donc il pointe sur le début de la chaîne.

La différence entre ces deux valeurs est maintenant calculée.

L'instruction SUB vient juste d'être effectuée:

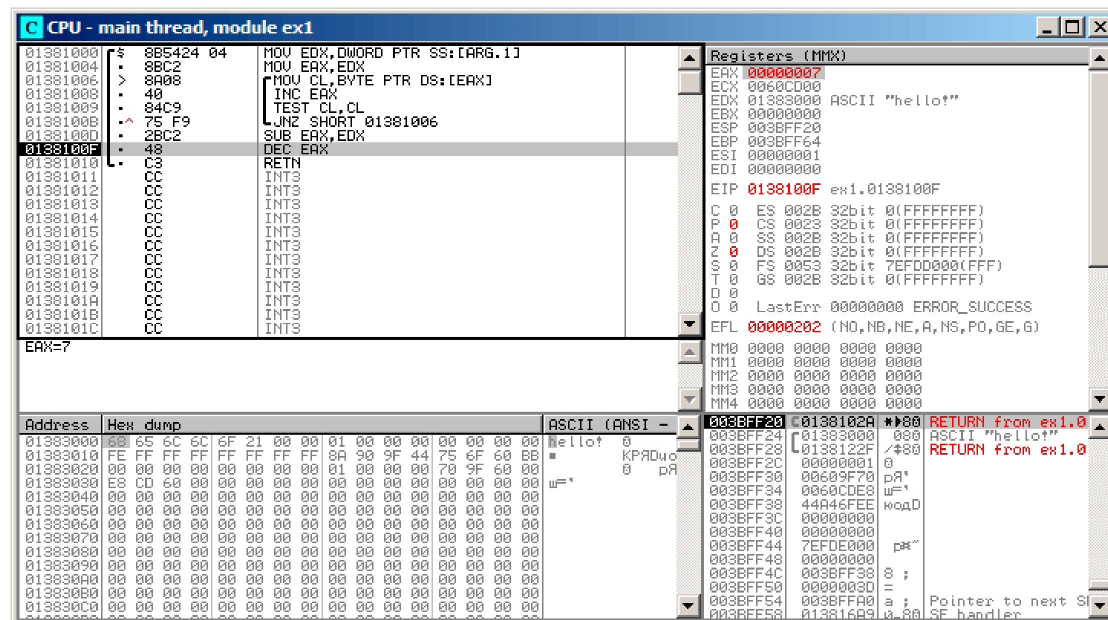


Fig. 1.61: OllyDbg : maintenant décrémenter EAX

La différence entre les deux pointeurs est maintenant dans le registre EAX—7. Effectivement, la longueur de la chaîne «hello! » est 6, mais avec l'octet à zéro inclus—7. Mais `strlen()` doit renvoyer le nombre de caractère non-zéro dans la chaîne. Donc la décrémentation est effectuée et ensuite la fonction sort.

GCC avec optimisation

Regardons ce que génère GCC 4.4.1 avec l'option d'optimisation -O3 :

```
                public strlen
strlen          proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     ecx, [ebp+arg_0]
                mov     eax, ecx

loc_8048418:
                movzx   edx, byte ptr [eax]
                add     eax, 1
                test    dl, dl
                jnz     short loc_8048418
```

```
                not     ecx
                add     eax, ecx
                pop     ebp
                retn
strlen          endp
```

mov dl, byte ptr [eax]. Ici GCC génère presque le même code que MSVC, à l'exception de la présence de MOVZX. Toutefois, ici, MOVZX pourrait être remplacé par mov dl, byte ptr [eax].

Peut-être est-il plus simple pour le générateur de code de GCC se se *rappeler* que le registre 32-bit EDX est alloué entièrement pour une variable *char* et il est sûr que les bits en partie haute ne contiennent pas de bruit indéfini.

Après cela, nous voyons une nouvelle instruction—NOT. Cette instruction inverse tout les bits de l'opérande.
Elle peut être vu comme un synonyme de l'instruction XOR ECX, 0ffffffffh. NOT et l'instruction suivante ADD calcule la différence entre les pointeurs et soustrait 1, d'une façon différente. Au début, ECX, où le pointeur sur *str* est stocké, est inversé et 1 en est soustrait.

En d'autres mots, à la fin de la fonction juste après le corps de la boucle, ces opérations sont exécutées:

```
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

… et ceci est effectivement équivalent à:

```
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

Pourquoi est-ce que GCC décide que cela est mieux? Difficile à deviner. Mais peut-être que les deux variantes sont également efficaces.

ARM

ARM 32-bit

sans optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Listing 1.189 : sans optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
_strlen
```

```
eos  = -8
str  = -4

     SUB    SP, SP, #8 ; allouer 8 octets pour les variables locales
     STR    R0, [SP,#8+str]
     LDR    R0, [SP,#8+str]
     STR    R0, [SP,#8+eos]

loc_2CB8 ; CODE XREF: _strlen+28
     LDR    R0, [SP,#8+eos]
     ADD    R1, R0, #1
     STR    R1, [SP,#8+eos]
     LDRSB  R0, [R0]
     CMP    R0, #0
     BEQ    loc_2CD4
     B      loc_2CB8
loc_2CD4 ; CODE XREF: _strlen+24
     LDR    R0, [SP,#8+eos]
     LDR    R1, [SP,#8+str]
     SUB    R0, R0, R1 ; R0=eos-str
     SUB    R0, R0, #1 ; R0=R0-1
     ADD    SP, SP, #8 ; libérer les 8 octets alloués
     BX     LR
```

LLVM sans optimisation génère beaucoup trop de code, toutefois, ici nous pouvons voir comment la fonction travaille avec les variables locales. Il y a seulement deux variables locales dans notre fonction: *eos* et *str*. Dans ce listing, généré par IDA, nous avons renommé manuellement *var_8* et *var_4* en *eos* et *str*.

La première instruction sauve simplement les valeurs d'entrée dans *str* et *eos*.

Le corps de la boucle démarre au label *loc_2CB8*.

Les trois première instructions du corps de la boucle (LDR, ADD, STR) chargent la valeur de *eos* dans R0. Puis la valeur est incrémentée et sauvée dans *eos*, qui se trouve sur la pile.

L'instruction suivante, LDRSB R0, [R0] («Load Register Signed Byte»), charge un octet depuis la mémoire à l'adresse stockée dans RR0 et étend le signe à 32-bit[105]. Ceci est similaire à l'instruction MOVSX en x86.

Le compilateur traite cet octet comme signé, puisque le type *char* est signé selon la norme C. Il a déjà été écrit à propos de cela (1.23.1 on page 264) dans cette section, en relation avec le x86.

Il est à noter qu'il est impossible en ARM d'utiliser séparément la partie 8- ou 16-bit d'un registre 32-bit complet, comme c'est le cas en x86.

Apparemment, c'est parce que le x86 à une énorme histoire de rétro-compatibilité avec ses ancêtres, jusqu'au 8086 16-bit et même 8080 8-bit, mais ARM a été développé à partir de zéro comme un processeur RISC 32-bit.

[105] Le compilateur Keil considère le type *char* comme signé, tout comme MSVC et GCC.

Par conséquent, pour manipuler des octets séparés en ARM, on doit tout de même utiliser des registres 32-bit.

Donc, LDRSB charge des octets depuis la chaîne vers R0, un par un. Les instructions suivantes, CMP et BEQ vérifient si l'octet chargé est 0. Si il n'est pas à 0, le contrôle passe au début du corps de la boucle. Et si c'est 0, la boucle est terminée.

À la fin de la fonction, la différence entre *eos* et *str* est calculée, 1 en est soustrait, et la valeur résultante est renvoyée via R0.

N.B. Les registres n'ont pas été sauvés dans cette fonction.

C'est parce que dans la convention d'appel ARM, les registres R0-R3 sont des «registres scratch», destinés à passer les arguments, et il n'est pas requis de restaurer leur valeur en sortant de la fonction, puisque la fonction appelante ne va plus les utiliser. Par conséquent, ils peuvent être utilisés comme bien nous semble.

Il n'y a pas d'autres registres utilisés ici, c'est pourquoi nous n'avons rien à sauvegarder sur la pile.

Ainsi, le contrôle peut être rendu à la fonction appelante par un simple saut (BX), à l'adresse contenue dans le registre LR.

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb)

Listing 1.190 : avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb)

```
_strlen
           MOV       R1, R0

loc_2DF6
           LDRB.W    R2, [R1],#1
           CMP       R2, #0
           BNE       loc_2DF6
           MVNS      R0, R0
           ADD       R0, R1
           BX        LR
```

Comme le conclut LLVM avec l'optimisation, *eos* et *str* n'ont pas besoin d'espace dans la pile, et peuvent toujours être stockés dans les registres.

Avant le début du corps de la boucle, *str* est toujours dans R0, et *eos*—dans R1.

L'instruction LDRB.W R2, [R1],#1 charge, dans R2, un octet de la mémoire à l'adresse stockée dans R1, en étendant le signe à une valeur 32-bit, mais pas seulement cela. #1 à la fin de l'instruction indique un «Adressage post-indexé» («Post-indexed addressing»), qui signifie que 1 doit être ajouté à R1 après avoir chargé l'octet. Pour en lire plus à ce propos: 1.39.2 on page 568.

Ensuite vous pouvez voir CMP et BNE[106] dans le corps de la boucle, ces instructions continuent de boucler jusqu'à ce que 0 soit trouvé dans la chaîne.

[106](PowerPC, ARM) Branch if Not Equal

Les instructions MVNS[107] (inverse tous les bits, comme NOT en x86) et ADD calculent $eos - str - 1$. (1.23.1 on page 271). En fait, ces deux instructions calculent $R0 = str + eos$, qui est effectivement équivalent à ce qui est dans le code source, et la raison de ceci à déjà été expliquée ici (1.23.1 on page 271).

Apparemment, LLVM, tout comme GCC, conclu que ce code peut être plus court (ou plus rapide).

avec optimisation Keil 6/2013 (Mode ARM)

Listing 1.191 : avec optimisation Keil 6/2013 (Mode ARM)

```
_strlen
                MOV     R1, R0

loc_2C8
                LDRB    R2, [R1],#1
                CMP     R2, #0
                SUBEQ   R0, R1, R0
                SUBEQ   R0, R0, #1
                BNE     loc_2C8
                BX      LR
```

Presque la même chose que ce que nous avions vu avant, à l'exception que l'expression $str - eos - 1$ peut être calculée non pas à la fin de la fonction, mais dans le corps de la boucle. Le suffixe -EQ, comme nous devrions nous en souvenir, implique que l'instruction ne s'exécute que si les opérandes de la dernière instruction CMP qui a été exécutée avant étaient égaux. Ainsi, si R0 contient 0, les deux instructions SUBEQ sont exécutées et le résultat est laissé dans le registre R0.

ARM64

GCC avec optimisation (Linaro) 4.9

```
my_strlen:
        mov     x1, x0
        ; X1 est maintenant un pointeur temporaire (eos), se comportant comme
    un curseur
.L58:
        ; charger un octet de X1 dans W2, incrémenter X1 (post-index)
        ldrb    w2, [x1],1
        ; Compare and Branch if NonZero: comparer W2 avec 0,
        ; sauter en .L58 si il ne l'est pas
        cbnz    w2, .L58
        ; calculer la différence entre le pointeur initial dans X0
        ; et l'adresse courante dans X1
        sub     x0, x1, x0
```

107 MoVe Not

```
        ; decrement lowest 32-bit of result
        sub     w0, w0, #1
        ret
```

L'algorithme est le même que dans 1.23.1 on page 266 : trouver un octet à zéro, calculer la différence antre les pointeurs et décrémenter le résultat de 1.size_t Quelques commentaires ont été ajouté par l'auteur de ce livre.

La seule différence notable est que cet exemple est un peu faux:
my_strlen() renvoie une valeur *int* 32-bit, tandis qu'elle devrait renvoyer un type size_t ou un autre type 64-bit.

La raison est que, théoriquement, strlen() peut-être appelée pour un énorme bloc de mémoire qui dépasse 4GB, donc elle doit être capable de renvoyer une valeur 64-bit sur une plate-forme 64-bit.

À cause de cette erreur, la dernière instruction SUB opère sur la partie 32-bit du registre, tandis que la pénultième instruction SUB travaille sur un registre 64-bit complet (elle calcule la différence entre les pointeurs).

C'est une erreur de l'auteur, il est mieux de la laisser ainsi, comme un exemple de ce à quoi ressemble le code dans un tel cas.

GCC sans optimisation (Linaro) 4.9

```
my_strlen:
; prologue de la fonction
        sub     sp, sp, #32
; le premier argument (str) va être stocké dans [sp,8]
        str     x0, [sp,8]
        ldr     x0, [sp,8]
; copier "str" dans la variable "eos"
        str     x0, [sp,24]
        nop
.L62:
; eos++
        ldr     x0, [sp,24]  ; charger "eos" dans X0
        add     x1, x0, 1    ; incrémenter X0
        str     x1, [sp,24]  ; sauver X0 dans "eos"
; charger dans W0 un octet de la mémoire à l'adresse dans X0
        ldrb    w0, [x0]
; est-ce zéro? (WZR est le registre 32-bit qui contient toujours zéro)
        cmp     w0, wzr
; sauter si différent de zéro (Branch Not Equal)
        bne     .L62
; octet à zéro trouvé. calculer maintenant la différence
; charger "eos" dans X1
        ldr     x1, [sp,24]
; charger "str" dans X0
        ldr     x0, [sp,8]
; calculer la différence
        sub     x0, x1, x0
```

```
; décrémenter le résultat
        sub     w0, w0, #1
; épilogue de la fonction
        add     sp, sp, 32
        ret
```

C’est plus verbeux. Les variables sont beaucoup manipulées vers et depuis la mémoire (pile locale). Il y a la même erreur ici: l’opération de décrémentation se produit sur la partie 32-bit du registre.

MIPS

Listing 1.192 : avec optimisation GCC 4.4.5 (IDA)

```
my_strlen:
; la variable "eos" sera toujours dans $v1:
                move    $v1, $a0

loc_4:
; charger l'octet à l'adresse dans "eos" dans $a1:
                lb      $a1, 0($v1)
                or      $at, $zero ; slot de délai de branchement, NOP
; si l'octet chargé n'est pas zéro, sauter en loc_4:
                bnez    $a1, loc_4
; incrémenter "eos" de toutes façons:
                addiu   $v1, 1 ; slot de délai de branchement
; boucle terminée. inverser variable "str":
                nor     $v0, $zero, $a0
; $v0=-str-1
                jr      $ra
; valeur de retour = $v1 + $v0 = eos + ( -str-1 ) = eos - str - 1
                addu    $v0, $v1, $v0 ; slot de délai de branchement
```

Il manque en MIPS une instruction `NOT`, mais il y a `NOR` qui correspond à l’opération `OR + NOT`.

Cette opération est largement utilisée en électronique digitale[108]. Par exemple, l’Apollo Guidance Computer (ordinateur de guidage Apollo) utilisé dans le programme Apollo, a été construit en utilisant seulement 5600 portes NOR: [Jens Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*, (2011)]. Mais l’élément NOT n’est pas très populaire en programmation informatique.

Donc, l’opération NOT est implémentée ici avec `NOR DST, $ZERO, SRC`.

D’après le chapitre sur les fondamentaux nous savons qu’une inversion des bits d’un nombre signé est la même chose que changer son signe et soustraire 1 du résultat.

Donc ce que `NOT` fait ici est de prendre la valeur de str et de la transformer en $-str-1$. L’opération d’addition qui suit prépare le résultat.

[108] NOR est appelé «porte universelle»

1.23.2 Limites de chaînes

Il est intéressant de noter comment les paramètres sont passés à la fonction win32 *GetOpenFileName()*. Afin de l'appeler, il faut définir une liste des extensions de fichier autorisées:

```
        OPENFILENAME *LPOPENFILENAME;
        ...
        char * filter = "Text files (*.txt)\0*.txt\0MS Word files (*.doc)↙
    ↘ \0*.doc\0\0";
        ...
        LPOPENFILENAME = (OPENFILENAME *)malloc(sizeof(OPENFILENAME));
        ...
        LPOPENFILENAME->lpstrFilter = filter;
        ...

        if(GetOpenFileName(LPOPENFILENAME))
        {
                ...
```

Ce qui se passe ici, c'est que la liste de chaînes est passée à *GetOpenFileName()*. Ce n'est pas un problème de l'analyser: à chaque fois que l'on rencontre un octet nul, c'est un élément. Quand on rencontre deux octets nul, c'est la fin de la liste. Si vous passez cette chaîne à `printf()`, elle traitera le premier élément comme une simple chaîne.

Donc, ceci est un chaîne, ou...? Il est plus juste de dire que c'est un buffer contenants plusieurs chaînes-C terminées par zéro, qui peut être stocké et traité comme un tout.

Un autre exemple est la fonction *strtok()*. Elle prend une chaîne et y écrit des octets nul. C'est ainsi qu'elle transforme la chaîne d'entrée en une sorte de buffer, qui contient plusieurs chaînes-C terminées par zéro.

1.24 Remplacement de certaines instructions arithmétiques par d'autres

Lors de la recherche d'optimisation, une instruction peut-être remplacée par une autre, ou même par un groupe d'instructions. Par exemple, ADD et SUB peuvent se remplacer: ligne 18 de listado.3.122.

Par exemple, l'instruction LEA est souvent utilisée pour des calculs arithmétiques simples: .1.6 on page 1333.

1.24.1 Multiplication

Multiplication en utilisant l'addition

Voici un exemple simple:

```
unsigned int f(unsigned int a)
{
```

```
        return a*8;
};
```

La multiplication par 8 a été remplacée par 3 instructions d'addition, qui font la même chose. Il semble que l'optimiseur de MSVC a décidé que ce code peut être plus rapide.

Listing 1.193 : MSVC 2010 avec optimisation

```
_TEXT   SEGMENT
_a$ = 8         ; size = 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, eax
        add     eax, eax
        add     eax, eax
        ret     0
_f      ENDP
_TEXT   ENDS
END
```

Multiplication en utilisant le décalage

Les instructions de multiplication et de division par un nombre qui est une puissance de 2 sont souvent remplacées par des instructions de décalage.

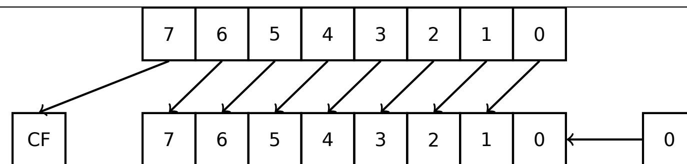
```
unsigned int f(unsigned int a)
{
        return a*4;
};
```

Listing 1.194 : MSVC 2010 sans optimisation

```
_a$ = 8         ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        shl     eax, 2
        pop     ebp
        ret     0
_f      ENDP
```

La multiplication par 4 consiste en un décalage du nombre de 2 bits vers la gauche et l'insertion de deux bits à zéro sur la droite (les deux derniers bits). C'est comme multiplier 3 par 100 —nous devons juste ajouter deux zéros sur la droite.

C'est ainsi que fonctionne l'instruction de décalage vers la gauche:



Les bits ajoutés à droite sont toujours des zéros.

Multiplication par 4 en ARM:

Listing 1.195 : sans optimisation Keil 6/2013 (Mode ARM)

```
f PROC
        LSL      r0,r0,#2
        BX       lr
        ENDP
```

Multiplication par 4 en MIPS:

Listing 1.196 : GCC 4.4.5 avec optimisation (IDA)

```
      jr      $ra
      sll     $v0, $a0, 2 ; branch delay slot
```

SLL signifie «Shift Left Logical» (décalage logique à gauche).

Multiplication en utilisant le décalage, la soustraction, et l'addition

Il est aussi possible de se passer des opérations de multiplication lorsque l'on multiplie par des nombres comme 7 ou 17, toujours en utilisant le décalage. Les mathématiques utilisées ici sont assez faciles.

32-bit

```
#include <stdint.h>

int f1(int a)
{
        return a*7;
};

int f2(int a)
{
        return a*28;
};

int f3(int a)
{
        return a*17;
};
```

x86

Listing 1.197 : MSVC 2012 avec optimisation

```
; a*7
_a$ = 8
_f1     PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        ret     0
_f1     ENDP

; a*28
_a$ = 8
_f2     PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        shl     eax, 2
; EAX=EAX<<2=(a*7)*4=a*28
        ret     0
_f2     ENDP

; a*17
_a$ = 8
_f3     PROC
        mov     eax, DWORD PTR _a$[esp-4]
; EAX=a
        shl     eax, 4
; EAX=EAX<<4=EAX*16=a*16
        add     eax, DWORD PTR _a$[esp-4]
; EAX=EAX+a=a*16+a=a*17
        ret     0
_f3     ENDP
```

ARM

Keil pour le mode ARM tire partie du décalage de registre du second opérande:

Listing 1.198 : avec optimisation Keil 6/2013 (Mode ARM)

```
; a*7
||f1|| PROC
        RSB      r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
```

```
        BX      lr
        ENDP

; a*28
||f2|| PROC
        RSB     r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
        LSL     r0,r0,#2
; R0=R0<<2=R0*4=a*7*4=a*28
        BX      lr
        ENDP

; a*17
||f3|| PROC
        ADD     r0,r0,r0,LSL #4
; R0=R0+R0<<4=R0+R0*16=R0*17=a*17
        BX      lr
        ENDP
```

Mais ce n'est pas disponible en mode Thumb. Il ne peut donc pas l'optimiser:

Listing 1.199 : avec optimisation Keil 6/2013 (Mode Thumb)

```
; a*7
||f1|| PROC
        LSLS    r1,r0,#3
; R1=R0<<3=a<<3=a*8
        SUBS    r0,r1,r0
; R0=R1-R0=a*8-a=a*7
        BX      lr
        ENDP

; a*28
||f2|| PROC
        MOVS    r1,#0x1c ; 28
; R1=28
        MULS    r0,r1,r0
; R0=R1*R0=28*a
        BX      lr
        ENDP

; a*17
||f3|| PROC
        LSLS    r1,r0,#4
; R1=R0<<4=R0*16=a*16
        ADDS    r0,r0,r1
; R0=R0+R1=a+a*16=a*17
        BX      lr
        ENDP
```

MIPS

Listing 1.200 : GCC 4.4.5 avec optimisation (IDA)

```
_f1:
                sll     $v0, $a0, 3
; $v0 = $a0<<3 = $a0*8
                jr      $ra
                subu    $v0, $a0 ; branch delay slot
; $v0 = $v0-$a0 = $a0*8-$a0 = $a0*7

_f2:
                sll     $v0, $a0, 5
; $v0 = $a0<<5 = $a0*32
                sll     $a0, 2
; $a0 = $a0<<2 = $a0*4
                jr      $ra
                subu    $v0, $a0 ; branch delay slot
; $v0 = $a0*32-$a0*4 = $a0*28

_f3:
                sll     $v0, $a0, 4
; $v0 = $a0<<4 = $a0*16
                jr      $ra
                addu    $v0, $a0 ; branch delay slot
; $v0 = $a0*16+$a0 = $a0*17
```

64-bit

```
#include <stdint.h>

int64_t f1(int64_t a)
{
        return a*7;
};

int64_t f2(int64_t a)
{
        return a*28;
};

int64_t f3(int64_t a)
{
        return a*17;
};
```

x64

Listing 1.201 : MSVC 2012 avec optimisation

```
; a*7
f1:
```

```
        lea     rax, [0+rdi*8]
; RAX=RDI*8=a*8
        sub     rax, rdi
; RAX=RAX-RDI=a*8-a=a*7
        ret

; a*28
f2:
        lea     rax, [0+rdi*4]
; RAX=RDI*4=a*4
        sal     rdi, 5
; RDI=RDI<<5=RDI*32=a*32
        sub     rdi, rax
; RDI=RDI-RAX=a*32-a*4=a*28
        mov     rax, rdi
        ret

; a*17
f3:
        mov     rax, rdi
        sal     rax, 4
; RAX=RAX<<4=a*16
        add     rax, rdi
; RAX=a*16+a=a*17
        ret
```

ARM64

GCC 4.9 pour ARM64 est aussi concis, grâce au modificateur de décalage:

Listing 1.202 : GCC (Linaro) 4.9 avec optimisation ARM64

```
; a*7
f1:
        lsl     x1, x0, 3
; X1=X0<<3=X0*8=a*8
        sub     x0, x1, x0
; X0=X1-X0=a*8-a=a*7
        ret

; a*28
f2:
        lsl     x1, x0, 5
; X1=X0<<5=a*32
        sub     x0, x1, x0, lsl 2
; X0=X1-X0<<2=a*32-a<<2=a*32-a*4=a*28
        ret

; a*17
f3:
        add     x0, x0, x0, lsl 4
; X0=X0+X0<<4=a+a*16=a*17
```

```
        ret
```

Algorithme de multiplication de Booth

Il fût un temps où les ordinateurs étaient si gros et chers, que certains d'entre eux ne disposaient pas de la multiplication dans le CPU, comme le Data General Nova. Et lorsque l'on avait besoin de l'opérateur de multiplication, il pouvait être fourni au niveau logiciel, par exemple, en utilisant l'algorithme de multiplication de Booth. C'est un algorithme de multiplication qui utilise seulement des opérations d'addition et de décalage.

Ce que les optimiseurs des compilateurs modernes font n'est pas la même chose, mais le but (multiplication) et les ressources (des opérations plus rapides) sont les mêmes.

1.24.2 Division

Division en utilisant des décalages

Exemple de division par 4:

```
unsigned int f(unsigned int a)
{
        return a/4;
};
```

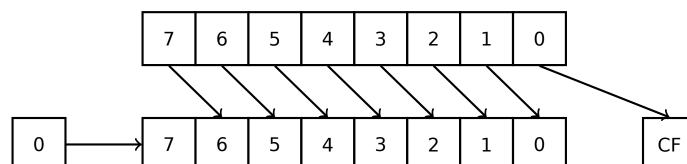
Nous obtenons (MSVC 2010) :

Listing 1.203 : MSVC 2010

```
_a$ = 8         ; size = 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        shr     eax, 2
        ret     0
_f      ENDP
```

L'instruction SHR (*SHift Right* décalage à droite) dans cet exemple décale un nombre de 2 bits vers la droite. Les deux bits libérés à gauche (i.e., les deux bits les plus significatifs) sont mis à zéro. Les deux bits les moins significatifs sont perdus. En fait, ces deux bits perdus sont le reste de la division.

L'instruction SHR fonctionne tout comme SHL, mais dans l'autre direction.



Il est facile de comprendre si vous imaginez le nombre 23 dans le système décimal. 23 peut être facilement divisé par 10, juste en supprimant le dernier chiffre (3—le reste de la division). Il reste 2 après l'opération, qui est le quotient.

Donc, le reste est perdu, mais c'est OK, nous travaillons de toutes façons sur des valeurs entières, ce sont sont pas des nombres réels !

Division par 4 en ARM:

Listing 1.204 : sans optimisation Keil 6/2013 (Mode ARM)

```
f PROC
        LSR      r0,r0,#2
        BX       lr
        ENDP
```

Division par 4 en MIPS:

Listing 1.205 : GCC 4.4.5 avec optimisation (IDA)

```
        jr      $ra
        srl     $v0, $a0, 2 ; slot de délai de branchement
```

L'instruction SLR est «Shift Right Logical» (décalage logique à droite).

1.24.3 Exercice

- http://challenges.re/59

1.25 Unité à virgule flottante

Le FPU est un dispositif à l'intérieur du CPU, spécialement conçu pour traiter les nombres à virgules flottantes.

Il était appelé «coprocesseur» dans le passé et il était en dehors du CPU.

1.25.1 IEEE 754

Un nombre au format IEEE 754 consiste en un *signe*, un *significande* (aussi appelé *fraction*) et un *exposant*.

1.25.2 x86

Ça vaut la peine de jeter un œil sur les machines à base de piles ou d'apprendre les bases du langage Forth, avant d'étudier le FPU en x86.

Il est intéressant de savoir que dans le passé (avant le CPU 80486) le coprocesseur était une puce séparée et n'était pas toujours préinstallé sur la carte mère. Il était possible de l'acheter séparément et de l'installer[109].

[109] Par exemple, John Carmack a utilisé des valeurs arithmétiques à virgule fixe dans son jeu vidéo Doom, stockées dans des registres 32-bit GPR (16 bit pour la partie entière et 16 bit pour la partie fractionnaire), donc Doom pouvait fonctionner sur des ordinateurs 32-bit sans FPU, i.e., 80386 et 80486 SX.

A partir du CPU 80486 DX, le FPU est intégré dans le CPU.

L'instruction FWAIT nous rappelle le fait qu'elle passe le CPU dans un état d'attente, jusqu'à ce que le FPU ait fini son traitement.

Un autre rudiment est le fait que les opcodes d'instruction FPU commencent avec ce qui est appelé l'opcode-«d'échappement» (D8..DF), i.e., opcodes passés à un coprocesseur séparé.

Le FPU a une pile capable de contenir 8 registres de 80-bit, et chaque registre peut contenir un nombre au format IEEE 754.

Ce sont ST(0)..ST(7). Par concision, IDA et OllyDbg montrent ST(0) comme ST, qui est représenté dans certains livres et manuels comme «Stack Top».

1.25.3 ARM, MIPS, x86/x64 SIMD

En ARM et MIPS le FPU n'a pas de pile, mais un ensemble de registres, qui peuvent être accédés aléatoirement, comme GPR.

La même idéologie est utilisée dans l'extension SIMD des CPUs x86/x64.

1.25.4 C/C++

Le standard des langages C/C++ offre au moins deux types de nombres à virgule flottante, *float* (*simple-précision*, 32 bits) [110] et *double* (*double-précision*, 64 bits).

Dans [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997)246] nous pouvons trouver que *simple-précision* signifie que la valeur flottante peut être stockée dans un simple mot machine [32-bit], *double-précision* signifie qu'elle peut être stockée dans deux mots (64 bits).

GCC supporte également le type *long double* (*précision étendue*, 80 bit), que MSVC ne supporte pas.

Le type *float* nécessite le même nombre de bits que le type *int* dans les environnements 32-bit, mais la représentation du nombre est complètement différente.

1.25.5 Exemple simple

Considérons cet exemple simple:

```
#include <stdio.h>

double f (double a, double b)
{
        return a/3.14 + b*4.1;
};

int main()
{
```

[110] le format des nombres à virgule flottante simple précision est aussi abordé dans la section *Travailler avec le type float comme une structure* (1.30.6 on page 481)

```
        printf ("%f\n", f(1.2, 3.4));
};
```

x86

MSVC

Compilons-le avec MSVC 2010:

Listing 1.206 : MSVC 2010: f()

```
CONST    SEGMENT
__real@4010666666666666 DQ 04010666666666666r    ; 4.1
CONST    ENDS
CONST    SEGMENT
__real@40091eb851eb851f DQ 040091eb851eb851fr    ; 3.14
CONST    ENDS
_TEXT    SEGMENT
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_f  PROC
    push   ebp
    mov    ebp, esp
    fld    QWORD PTR _a$[ebp]

; état courant de la pile: ST(0) = _a

    fdiv   QWORD PTR __real@40091eb851eb851f

; état courant de la pile: ST(0) = résultat de _a divisé par 3.14

    fld    QWORD PTR _b$[ebp]

; état courant de la pile: ST(0) = _b;
; ST(1) = résultat de _a divisé par 3.14

    fmul   QWORD PTR __real@4010666666666666

; état courant de la pile:
; ST(0) = résultat de _b * 4.1;
; ST(1) = résultat de _a divisé par 3.14

    faddp  ST(1), ST(0)

; état courant de la pile: ST(0) = résultat de l'addition

    pop    ebp
    ret    0
_f  ENDP
```

FLD prend 8 octets depuis la pile et charge le nombre dans le registre ST(0), en

le convertissant automatiquement dans le format interne sur 80-bit (*précision étendue*) :

FDIV divise la valeur dans ST(0) par le nombre stocké à l'adresse __real@40091eb851eb851f —la valeur 3.14 est encodée ici. La syntaxe assembleur ne supporte pas les nombres à virgule flottante, donc ce que l'on voit ici est la représentation hexadécimale de 3.14 au format 64-bit IEEE 754.

Après l'exécution de FDIV, ST(0) contient le quotient.

À propos, il y a aussi l'instruction FDIVP, qui divise ST(1) par ST(0), prenant ces deux valeurs dans la pile et poussant le résultant. Si vous connaissez le langage Forth, vous pouvez comprendre rapidement que ceci est une machine à pile.

L'instruction FLD subséquente pousse la valeur de b sur la pile.

Après cela, le quotient est placé dans ST(1), et ST(0) a la valeur de b.

L'instruction suivante effectue la multiplication: b de ST(0) est multiplié par la valeur en
__real@4010666666666666 (le nombre 4.1 est là) et met le résultat dans le registre ST(0).

La dernière instruction FADDP ajoute les deux valeurs au sommet de la pile, stockant le résultat dans ST(1) et supprimant la valeur de ST(0), laissant ainsi le résultat au sommet de la pile, dans ST(0).

La fonction doit renvoyer son résultat dans le registre ST(0), donc il n'y a aucune autre instruction après FADDP, excepté l'épilogue de la fonction.

2 paires de mots 32-bit sont marquées en rouge sur la pile. Chaque paire est un double au format IEEE 754 et est passée depuis `main()`.

Nous voyons comment le premier FLD charge une valeur (1.2) depuis la pile et la stocke dans `ST(0)` :

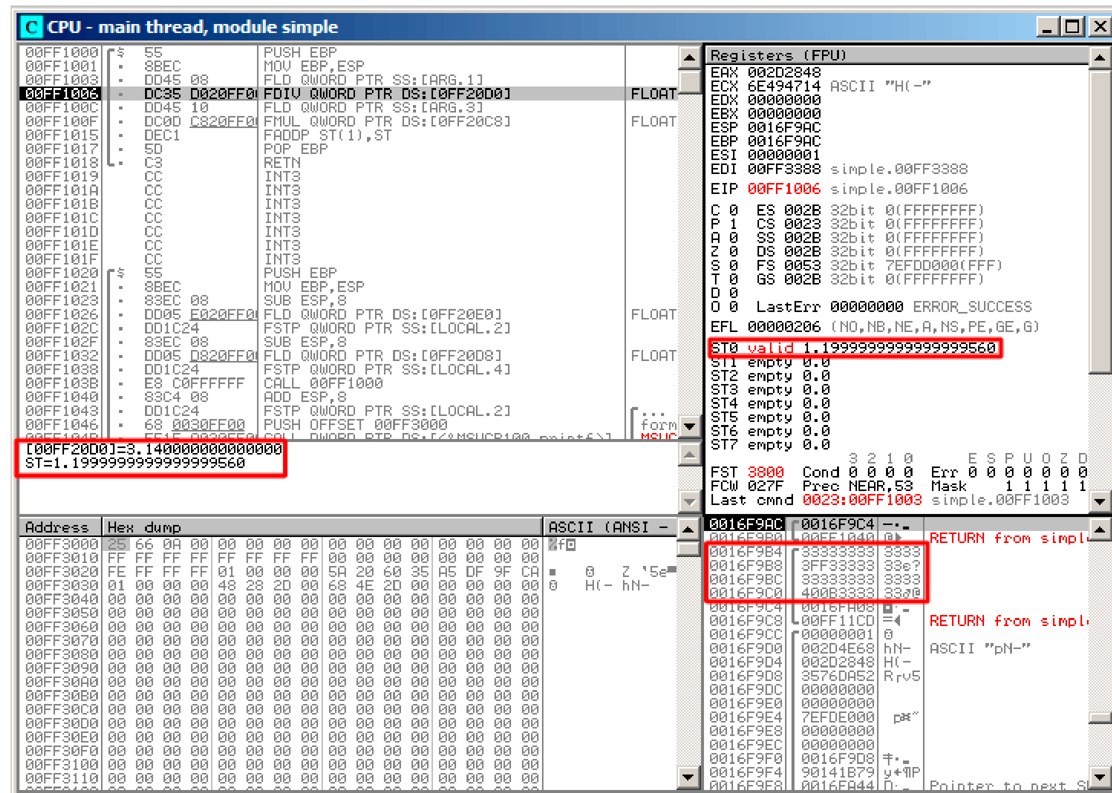


Fig. 1.62: OllyDbg : le premier FLD a été exécuté

À cause des inévitables erreurs de conversion depuis un nombre flottant 64-bit au format IEEE 754 vers 80-bit (utilisé en interne par le FPU), ici nous voyons 1.199…, qui est proche de 1.2.

EIP pointe maintenant sur l'instruction suivante (FDIV), qui charge un double (une constante) depuis la mémoire. Par commodité, OllyDbg affiche sa valeur: 3.14

Continuons l'exécution pas à pas. FDIV a été exécuté, maintenant ST(0) contient 0.382…(quotient) :

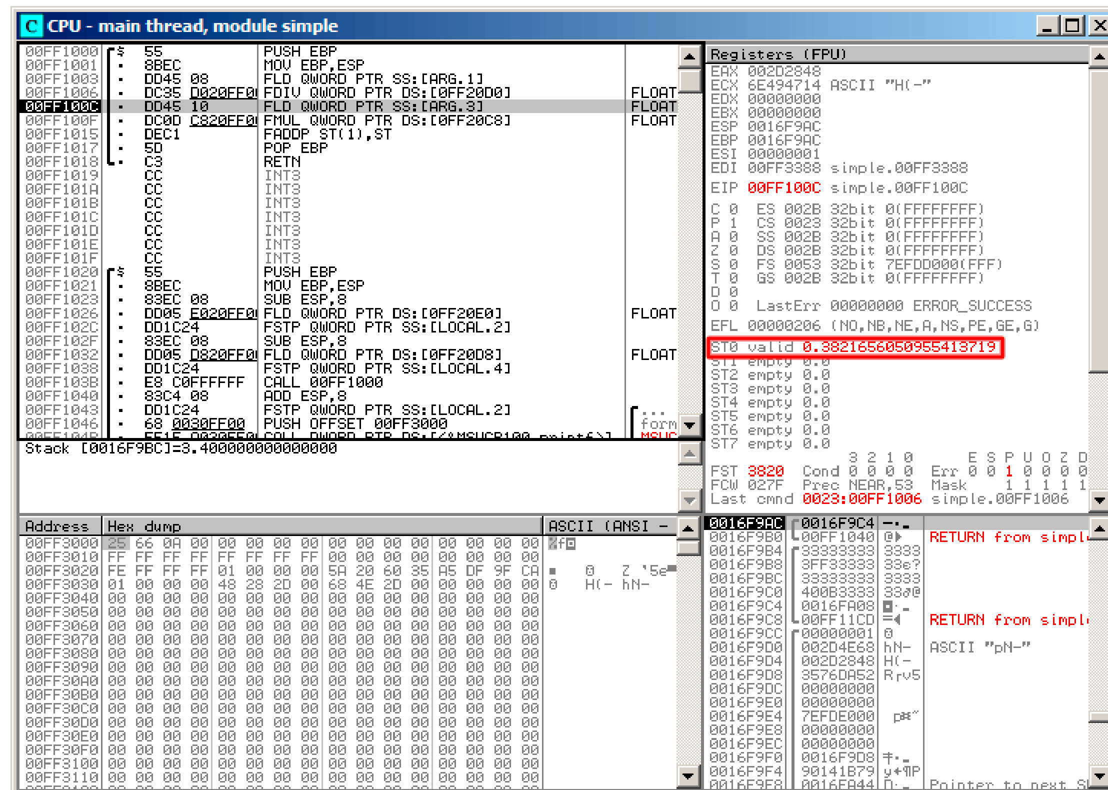


Fig. 1.63: OllyDbg : FDIV a été exécuté

Troisième étape: le FLD suivant a été exécuté, chargeant 3.4 dans ST(0) (ici nous voyons la valeur approximative 3.39999...) :

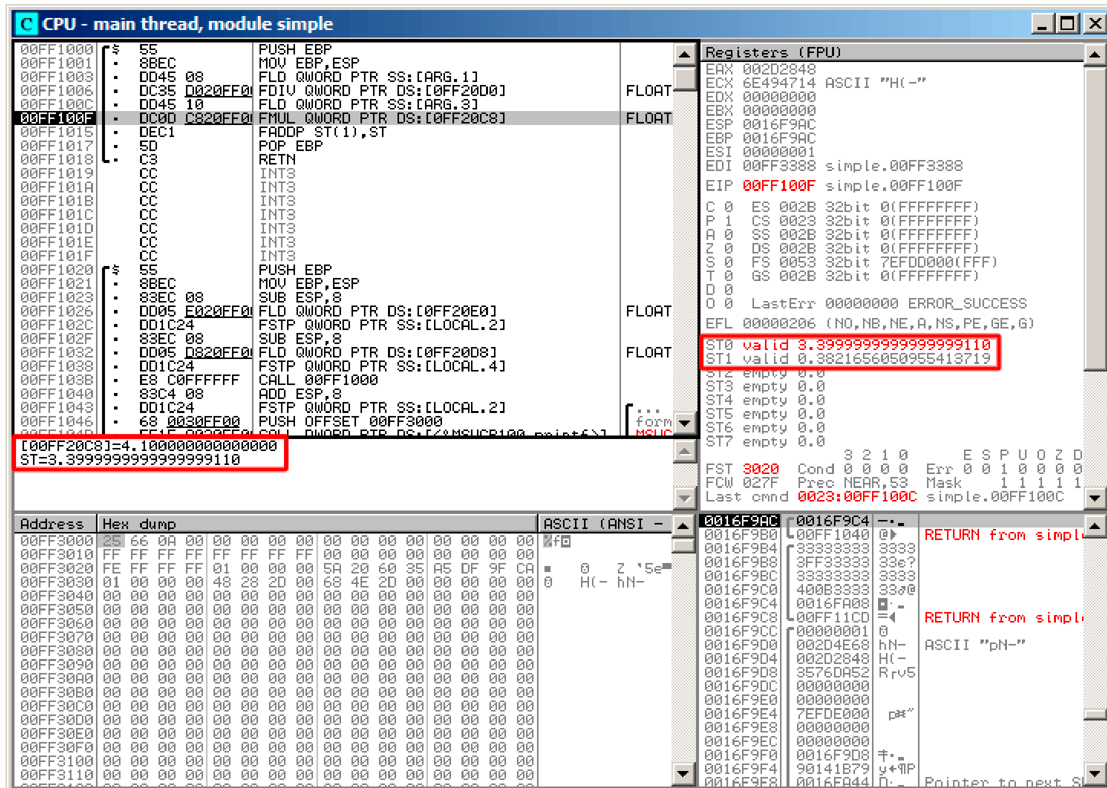


Fig. 1.64: OllyDbg : le second FLD a été exécuté

En même temps, le quotient *est poussé* dans ST(1). Exactement maintenant, EIP pointe sur la prochaine instruction: FMUL. Ceci charge la constante 4.1 depuis la mémoire, ce que montre OllyDbg.

Suivante: FMUL a été exécutée, donc maintenant le produit est dans ST(0) :

Fig. 1.65: OllyDbg : FMUL a été exécuté

Suivante: FADDP a été exécutée, maintenant le résultat de l'addition est dans `ST(0)`, et `ST(1)` est vidé.

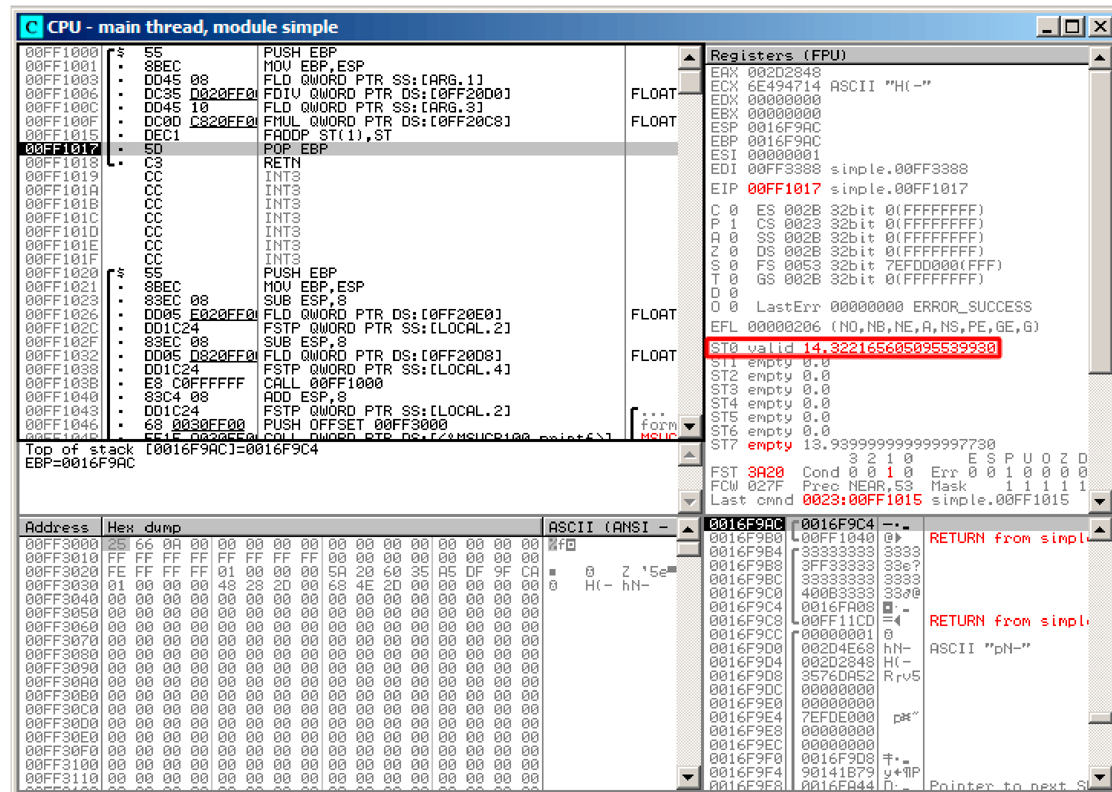


Fig. 1.66: OllyDbg : FADDP a été exécuté

Le résultat est laissé dans `ST(0)`, car la fonction renvoie son résultat dans `ST(0)`.

`main()` prend cette valeur depuis le registre plus loin.

Nous voyons quelque chose d'inhabituel: la valeur 13.93...se trouve maintenant dans `ST(7)`. Pourquoi?

Comme nous l'avons lu il y a quelque temps dans ce livre, les registres FPU sont une pile: 1.25.2 on page 286. Mais ceci est une simplification.

Imaginez si cela était implémenté *en hardware* comme cela est décrit, alors tout le contenu des 7 registres devrait être déplacé (ou copié) dans les registres adjacents lors d'un push ou d'un pop, et ceci nécessite beaucoup de travail.

En réalité, le FPU a seulement 8 registres et un pointeur (appelé TOP) qui contient un numéro de registre, qui est le «haut de la pile» courant.

Lorsqu'une valeur est poussée sur la pile, TOP est déplacé sur le registre disponible suivant, et une valeur est écrite dans ce registre.

La procédure est inversée si la valeur est lue, toutefois, le registre qui a été libéré n'est pas vidé (il serait possible de le vider, mais ceci nécessite plus de travail qui peut dégrader les performances). Donc, c'est ce que nous voyons ici.

On peut dire que FADDP sauve la somme sur la pile, et y supprime un élément.

Mais en fait, cette instruction sauve la somme et ensuite décale TOP.

Plus précisément, les registres du FPU sont un tampon circulaire.

GCC

GCC 4.4.1 (avec l'option -O3) génère le même code, juste un peu différent:

Listing 1.207 : GCC 4.4.1 avec optimisation

```
                public f
f               proc near

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                fld     ds:dbl_8048608 ; 3.14

; état de la pile maintenant: ST(0) = 3.14

                mov     ebp, esp
                fdivr   [ebp+arg_0]

; état de la pile maintenant: ST(0) = résultat de la division

                fld     ds:dbl_8048610 ; 4.1

; état de la pile maintenant: ST(0) = 4.1, ST(1) = résultat de la division

                fmul    [ebp+arg_8]

; état de la pile maintenant: ST(0) = résultat de la multiplication, ST(1) =
    résultat de la division

                pop     ebp
                faddp   st(1), st

; état de la pile maintenant: ST(0) = résultat de l'addition

                retn
f               endp
```

La différence est que, tout d'abord, 3.14 est poussé sur la pile (dans ST(0)), et ensuite la valeur dans arg_0 est divisée par la valeur dans ST(0).

FDIVR signifie *Reverse Divide* —pour diviser avec le diviseur et le dividende échangés l'un avec l'autre. Il n'y a pas d'instruction de ce genre pour la multiplication

puisque c'est une opération commutative, donc nous avons seulement FMUL sans son homologue -R.

FADDP ajoute les deux valeurs mais supprime aussi une valeur de la pile. Après cette opération, ST(0) contient la somme.

ARM: avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Jusqu'à la standardisation du support de la virgule flottante, certains fabricants de processeur ont ajouté leur propre instructions étendues. Ensuite, VFP (*Vector Floating Point*) a été standardisé.

Une différence importante par rapport au x86 est qu'en ARM, il n'y a pas de pile, vous travaillez seulement avec des registres.

Listing 1.208 : avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
f
                VLDR            D16, =3.14
                VMOV            D17, R0, R1 ; charge "a"
                VMOV            D18, R2, R3 ; charge "b"
                VDIV.F64        D16, D17, D16 ; a/3.14
                VLDR            D17, =4.1
                VMUL.F64        D17, D18, D17 ; b*4.1
                VADD.F64        D16, D17, D16 ; +
                VMOV            R0, R1, D16
                BX              LR

dbl_2C98        DCFD 3.14               ; DATA XREF: f
dbl_2CA0        DCFD 4.1                ; DATA XREF: f+10
```

Donc, nous voyons ici que des nouveaux registres sont utilisés, avec le préfixe D.

Ce sont des registres 64-bits, il y en a 32, et ils peuvent être utilisés tant pour des nombres à virgules flottantes (double) que pour des opérations SIMD (c'est appelé NEON ici en ARM).

Il y a aussi 32 S-registres 32 bits, destinés à être utilisés pour les nombres à virgules flottantes simple précision (float).

C'est facile à retenir: les registres D sont pour les nombres en double précision, tandis que les registres S—-pour les nombres en simple précision Pour aller plus loin: .2.3 on page 1350.

Les deux constantes (3.14 et 4.1) sont stockées en mémoire au format IEEE 754.

VLDR et VMOV, comme il peut en être facilement déduit, sont analogues aux instructions LDR et MOV, mais travaillent avec des registres D.

Il est à noter que ces instructions, tout comme les registres D, sont destinées non seulement pour les nombres à virgules flottantes, mais peuvent aussi être utilisées pour des opérations SIMD (NEON) et cela va être montré bientôt.

Les arguments sont passés à la fonction de manière classique, via les R-registres, toutefois, chaque nombre en double précision a une taille de 64 bits, donc deux R-registres sont nécessaires pour passer chacun d'entre eux.

VMOV D17, R0, R1 au début, combine les deux valeurs 32-bit de R0 et R1 en une valeur 64-bit et la sauve dans D17.

VMOV R0, R1, D16 est l'opération inverse: ce qui est dans D16 est séparé dans deux registres, R0 et R1, car un nombre en double précision qui nécessite 64 bit pour le stockage, est renvoyé dans R0 et R1.

VDIV, VMUL and VADD, sont des instructions pour traiter des nombres à virgule flottante, qui calculent respectivement le quotient, produit et la somme.

Le code pour Thumb-2 est similaire.

ARM: avec optimisation Keil 6/2013 (Mode Thumb)

```
f
                PUSH    {R3-R7,LR}
                MOVS    R7, R2
                MOVS    R4, R3
                MOVS    R5, R0
                MOVS    R6, R1
                LDR     R2, =0x66666666 ; 4.1
                LDR     R3, =0x40106666
                MOVS    R0, R7
                MOVS    R1, R4
                BL      __aeabi_dmul
                MOVS    R7, R0
                MOVS    R4, R1
                LDR     R2, =0x51EB851F ; 3.14
                LDR     R3, =0x40091EB8
                MOVS    R0, R5
                MOVS    R1, R6
                BL      __aeabi_ddiv
                MOVS    R2, R7
                MOVS    R3, R4
                BL      __aeabi_dadd
                POP     {R3-R7,PC}

; 4.1 au format IEEE 754:
dword_364       DCD 0x66666666          ; DATA XREF: f+A
dword_368       DCD 0x40106666          ; DATA XREF: f+C
; 3.14 au format IEEE 754:
dword_36C       DCD 0x51EB851F          ; DATA XREF: f+1A
dword_370       DCD 0x40091EB8          ; DATA XREF: f+1C
```

Code généré par Keil pour un processeur sans FPU ou support pour NEON.

Les nombres en virgule flottante double précision sont passés par des R-registres génériques et au lieu d'instructions FPU, des fonctions d'une bibliothèque de service sont appelées (comme __aeabi_dmul, __aeabi_ddiv, __aeabi_dadd) qui émulent la multiplication, la division et l'addition pour les nombres à virgule flottante.

Bien sûr, c'est plus lent qu'un coprocesseur FPU, mais toujours mieux que rien.

À propos, de telles bibliothèques d'émulation de FPU étaient très populaires dans le monde x86 lorsque les coprocesseurs étaient rares et chers, et étaient installés seulement dans des ordinateurs coûteux.

L'émulation d'un coprocesseur FPU est appelée *soft float* ou *armel* (*emulation*) dans le monde ARM, alors que l'utilisation des instructions d'un coprocesseur FPU est appelée *hard float* ou *armhf*.

ARM64: GCC avec optimisation (Linaro) 4.9

Code très compact:

Listing 1.209 : GCC avec optimisation (Linaro) 4.9

```
f:
; D0 = a, D1 = b
        ldr     d2, .LC25       ; 3.14
; D2 = 3.14
        fdiv    d0, d0, d2
; D0 = D0/D2 = a/3.14
        ldr     d2, .LC26       ; 4.1
; D2 = 4.1
        fmadd   d0, d1, d2, d0
; D0 = D1*D2+D0 = b*4.1+a/3.14
        ret

; constantes au format IEEE 754:
.LC25:
        .word   1374389535      ; 3.14
        .word   1074339512
.LC26:
        .word   1717986918      ; 4.1
        .word   1074816614
```

ARM64: GCC sans optimisation (Linaro) 4.9

Listing 1.210 : GCC sans optimisation (Linaro) 4.9

```
f:
        sub     sp, sp, #16
        str     d0, [sp,8]      ; sauve "a" dans le Register Save Area
        str     d1, [sp]        ; sauve "b" dans le Register Save Area
        ldr     x1, [sp,8]
; X1 = a
        ldr     x0, .LC25
; X0 = 3.14
        fmov    d0, x1
        fmov    d1, x0
; D0 = a, D1 = 3.14
        fdiv    d0, d0, d1
; D0 = D0/D1 = a/3.14

        fmov    x1, d0
```

```
; X1 = a/3.14
        ldr     x2, [sp]
; X2 = b
        ldr     x0, .LC26
; X0 = 4.1
        fmov    d0, x2
; D0 = b
        fmov    d1, x0
; D1 = 4.1
        fmul    d0, d0, d1
; D0 = D0*D1 = b*4.1

        fmov    x0, d0
; X0 = D0 = b*4.1
        fmov    d0, x1
; D0 = a/3.14
        fmov    d1, x0
; D1 = X0 = b*4.1
        fadd    d0, d0, d1
; D0 = D0+D1 = a/3.14 + b*4.1

        fmov    x0, d0 ; \ code redondant
        fmov    d0, x0 ; /
        add     sp, sp, 16
        ret
.LC25:
        .word   1374389535      ; 3.14
        .word   1074339512
.LC26:
        .word   1717986918      ; 4.1
        .word   1074816614
```

GCC sans optimisation est plus verbeux.

Il y a des nombreuses modifications de valeur inutiles, incluant du code clairement redondant (les deux dernières instructions FMOV). Sans doute que GCC 4.9 n'est pas encore très bon pour la génération de code ARM64.

Il est utile de noter qu'ARM64 possède des registres 64-bit, et que les D-registres sont aussi 64-bit.

Donc le compilateur est libre de sauver des valeurs de type *double* dans GPRs au lieu de la pile locale. Ce n'est pas possible sur des CPUs 32-bit.

Et encore, à titre d'exercice, vous pouvez essayer d'optimiser manuellement cette fonction, sans introduire de nouvelles instructions comme FMADD.

1.25.6 Passage de nombres en virgule flottante par les arguments

```
#include <math.h>
#include <stdio.h>
```

```
int main ()
{
        printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

        return 0;
}
```

x86

Regardons ce que nous obtenons avec MSVC 2010:

Listing 1.211 : MSVC 2010

```
CONST    SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r    ; 1.54
CONST    ENDS

_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 8  ; allouer de l'espace pour la première variable
    fld    QWORD PTR __real@3ff8a3d70a3d70a4
    fstp   QWORD PTR [esp]
    sub    esp, 8  ; allouer de l'espace pour la seconde variable
    fld    QWORD PTR __real@40400147ae147ae1
    fstp   QWORD PTR [esp]
    call   _pow
    add    esp, 8  ; rendre l'espace d'une variable.

; sur la pile locale, il y a ici encore 8 octets réservés pour nous.
; le résultat se trouve maintenant dans ST(0)

; déplace le résultat de ST(0) vers la pile locale pour printf():
    fstp   QWORD PTR [esp]
    push   OFFSET $SG2651
    call   _printf
    add    esp, 12
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
```

FLD et FSTP déplacent des variables entre le segment de données et la pile du FPU. `pow()`[111] prend deux valeurs depuis la pile et renvoie son résultat dans le registre `ST(0)`. `printf()` prend 8 octets de la pile locale et les interprète comme des variables de type *double*.

À propos, une paire d'instructions `MOV` pourrait être utilisée ici pour déplacer les valeurs depuis la mémoire vers la pile, car les valeurs en mémoire sont stockées au

[111] une fonction C standard, qui élève un nombre à la puissance donnée (puissance)

format IEEE 754, et pow() les prend aussi dans ce format, donc aucune conversion n'est nécessaire. C'est fait ainsi dans l'exemple suivant, pour ARM: 1.25.6.

ARM + sans optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
_main

var_C     = -0xC

          PUSH     {R7,LR}
          MOV      R7, SP
          SUB      SP, SP, #4
          VLDR     D16, =32.01
          VMOV     R0, R1, D16
          VLDR     D16, =1.54
          VMOV     R2, R3, D16
          BLX      _pow
          VMOV     D16, R0, R1
          MOV      R0, 0xFC1 ; "32.01 ^ 1.54 = %lf\n"
          ADD      R0, PC
          VMOV     R1, R2, D16
          BLX      _printf
          MOVS     R1, 0
          STR      R0, [SP,#0xC+var_C]
          MOV      R0, R1
          ADD      SP, SP, #4
          POP      {R7,PC}

dbl_2F90  DCFD 32.01       ; DATA XREF: _main+6
dbl_2F98  DCFD 1.54        ; DATA XREF: _main+E
```

Comme nous l'avons déjà mentionné, les pointeurs sur des nombres flottants 64-bit sont passés dans une paire de R-registres.

Ce code est un peu redondant (probablement car l'optimisation est désactivée), puisqu'il est possible de charger les valeurs directement dans les R-registres sans toucher les D-registres.

Donc, comme nous le voyons, la fonction _pow reçoit son premier argument dans R0 et R1, et le second dans R2 et R3. La fonction laisse son résultat dans R0 et R1. Le résultat de _pow est déplacé dans D16, puis dans la paire R1 et R2, d'où printf() prend le nombre résultant.

ARM + sans optimisation Keil 6/2013 (Mode ARM)

```
_main
      STMFD   SP!, {R4-R6,LR}
      LDR     R2, =0xA3D70A4  ; y
      LDR     R3, =0x3FF8A3D7
      LDR     R0, =0xAE147AE1 ; x
      LDR     R1, =0x40400147
      BL      pow
```

```
        MOV     R4, R0
        MOV     R2, R4
        MOV     R3, R1
        ADR     R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\n"
        BL      __2printf
        MOV     R0, #0
        LDMFD   SP!, {R4-R6,PC}

y               DCD 0xA3D70A4           ; DATA XREF: _main+4
dword_520       DCD 0x3FF8A3D7          ; DATA XREF: _main+8
x               DCD 0xAE147AE1          ; DATA XREF: _main+C
dword_528       DCD 0x40400147          ; DATA XREF: _main+10
a32_011_54Lf    DCB "32.01 ^ 1.54 = %lf",0xA,0
                                        ; DATA XREF: _main+24
```

Les D-registres ne sont pas utilisés ici, juste des paires de R-registres.

ARM64 + GCC (Linaro) 4.9 avec optimisation

Listing 1.212 : GCC (Linaro) 4.9 avec optimisation

```
f:
        stp     x29, x30, [sp, -16]!
        add     x29, sp, 0
        ldr     d1, .LC1 ; charger 1.54 dans D1
        ldr     d0, .LC0 ; charger 32.01 dans D0
        bl      pow
; résultat de pow() dans D0
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret
.LC0:
; 32.01 au format IEEE 754
        .word   -1374389535
        .word   1077936455
.LC1:
; 1.54 au format IEEE 754
        .word   171798692
        .word   1073259479
.LC2:
        .string "32.01 ^ 1.54 = %lf\n"
```

Les constantes sont chargées dans D0 et D1 : pow() les prend d'ici. Le résultat sera dans D0 après l'exécution de pow(). Il est passé à printf() sans aucune modification ni déplacement, car printf() prend ces arguments de type intégral et pointeurs depuis des X-registres, et les arguments en virgule flottante depuis des D-registres.

1.25.7 Exemple de comparaison

```
#include <stdio.h>

double d_max (double a, double b)
{
        if (a>b)
                return a;

        return b;
};

int main()
{
        printf ("%f\n", d_max (1.2, 3.4));
        printf ("%f\n", d_max (5.6, -4));
};
```

Malgré la simplicité de la fonction, il va être difficile de comprendre comment elle fonctionne.

x86

MSVC sans optimisation

MSVC 2010 génère ce qui suit:

Listing 1.213 : MSVC 2010 sans optimisation

```
PUBLIC    _d_max
_TEXT    SEGMENT
_a$ = 8                ; size = 8
_b$ = 16               ; size = 8
_d_max     PROC
    push   ebp
    mov    ebp, esp
    fld    QWORD PTR _b$[ebp]

; état courant de la pile: ST(0) = _b
; comparer _b (ST(0)) et _a, et dépiler un registre

    fcomp  QWORD PTR _a$[ebp]

; la pile est vide ici

    fnstsw ax
    test   ah, 5
    jp     SHORT $LN1@d_max

; nous sommes ici seulement si if a>b

    fld    QWORD PTR _a$[ebp]
    jmp    SHORT $LN2@d_max
$LN1@d_max:
```

```
    fld     QWORD PTR _b$[ebp]
$LN2@d_max:
    pop     ebp
    ret     0
_d_max      ENDP
```

Ainsi, FLD charge _b dans ST(0).

FCOMP compare la valeur dans ST(0) avec ce qui est dans _a et met les bits C3/C2/C0 du mot registre d'état du FPU, suivant le résultat. Ceci est un registre 16-bit qui reflète l'état courant du FPU.

Après que les bits ont été mis, l'instruction FCOMP dépile une variable depuis la pile. C'est ce qui la différencie de FCOM, qui compare juste les valeurs, laissant la pile dans le même état.

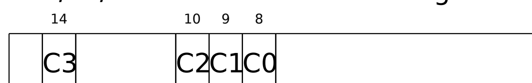
Malheureusement, les CPUs avant les Intel P6[112] ne possèdent aucune instruction de saut conditionnel qui teste les bits C3/C2/C0. Peut-être est-ce une raison historique (rappel: le FPU était une puce séparée dans le passé). Les CPU modernes, à partir des Intel P6 possèdent les instructions
FCOMI/FCOMIP/FUCOMI/FUCOMIP —qui font la même chose, mais modifient les flags ZF/PF/CF du CPU.

L'instruction FNSTSW copie le le mot du registre d'état du FPU dans AX. Les bits C3/C2/C0 sont placés aux positions 14/10/8, ils sont à la même position dans le registre AX et tous sont placés dans la partie haute de AX —AH.

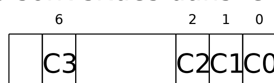
- Si $b > a$ dans notre exemple, alors les bits C3/C2/C0 sont mis comme ceci: 0, 0, 0.
- Si $a > b$, alors les bits sont: 0, 0, 1.
- Si $a = b$, alors les bits sont: 1, 0, 0.

  Si le résultat n'est pas ordonné (en cas d'erreur), alors les bits sont: 1, 1, 1.

Voici comment les bits C3/C2/C0 sont situés dans le registre AX :



Voici comment les bits C3/C2/C0 sont situés dans le registre AH :



Après l'exécution de `test ah, 5`[113], seul les bits C0 et C2 (en position 0 et 2) sont considérés, tous les autres bits sont simplement ignorés.

Parlons maintenant du *parity flag* (flag de parité), un autre rudiment historique remarquable.

Ce flag est mis à 1 si le nombre de un dans le résultat du dernier calcul est pair, et à 0 s'il est impair.

[112] Intel P6 comprend les Pentium Pro, Pentium II, etc.
[113] 5=101b

Regardons sur Wikipédia[114] :

> Une raison commune de tester le bit de parité n'a rien à voir avec la parité. Le FPU possède quatre flags de condition (C0 à C3), mais ils ne peuvent pas être testés directement, et doivent d'abord être copiés dans le registre d'états. Lorsque ça se produit, C0 est mis dans le flag de retenue, C2 dans le flag de parité et C3 dans le flag de zéro. Le flag C2 est mis lorsque e.g. des valeurs en virgule flottantes incomparable (NaN ou format non supporté) sont comparées avec l'instruction FUCOM.

Comme indiqué dans Wikipédia, le flag de parité est parfois utilisé dans du code FPU, voyons comment.

Le flag PF est mis à 1 si à la fois C0 et C2 sont mis à 0 ou si les deux sont à 1, auquel cas le JP (*jump if PF==1*) subséquent est déclenché. Si l'on se rappelle les valeurs de C3/C2/C0 pour différents cas, nous pouvons voir que le saut conditionnel JP est déclenché dans deux cas: si $b > a$ ou $a = b$ (le bit C3 n'est pris en considération ici, puisqu'il a été mis à 0 par l'instruction `test ah, 5`).

C'est très simple ensuite. Si le saut conditionnel a été déclenché, FLD charge la valeur de _b dans ST(0), et sinon, la valeur de _a est chargée ici.

Et à propos du test de C2 ?

Le flag C2 est mis en cas d'erreur (NaN, etc.), mais notre code ne le teste pas.

Si le programmeur veut prendre en compte les erreurs FPU, il doit ajouter des tests supplémentaires.

[114] https://en.wikipedia.org/wiki/Parity_flag

Premier exemple sous OllyDbg : a=1.2 et b=3.4

Chargeons l'exemple dans OllyDbg :

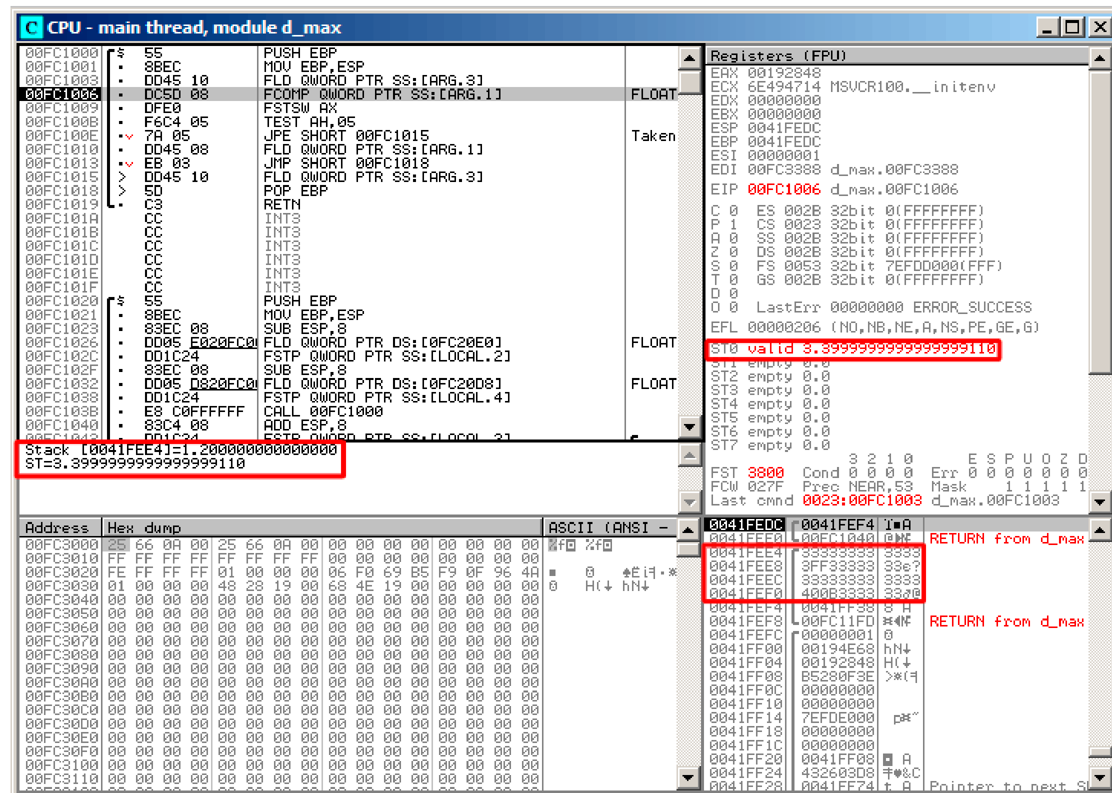


Fig. 1.67: OllyDbg : la première instruction FLD a été exécutée

Arguments courants de la fonction: $a = 1.2$ et $b = 3.4$ (Nous pouvons les voir dans la pile: deux paires de valeurs 32-bit). b (3.4) est déjà chargé dans ST(0). Maintenant FCOMP est train d'être exécutée. OllyDbg montre le second argument de FCOMP, qui se trouve sur la pile à ce moment.

FCOMP a été exécutée:

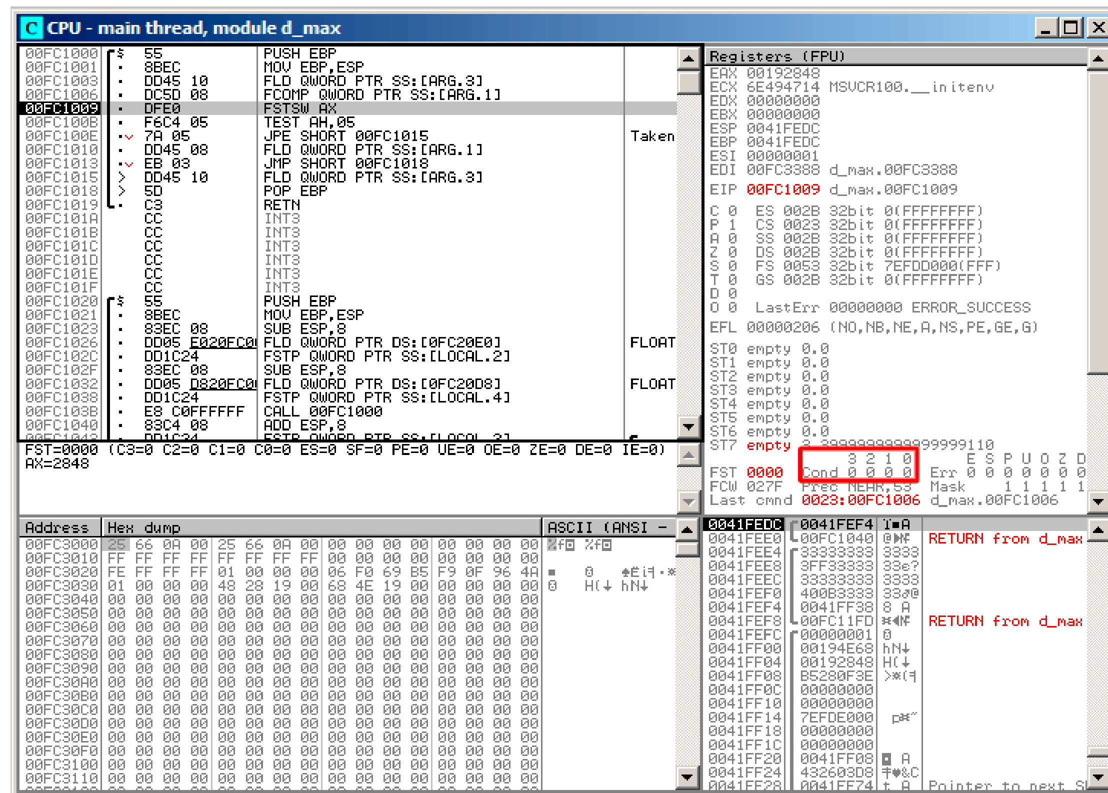


Fig. 1.68: OllyDbg : FCOMP a été exécutée

Nous voyons l'état des flags de condition du FPU : tous à zéro. La valeur dépilée est vue ici comme ST(7), la raison a été décrite ici: 1.25.5 on page 293.

FNSTSW a été exécutée:

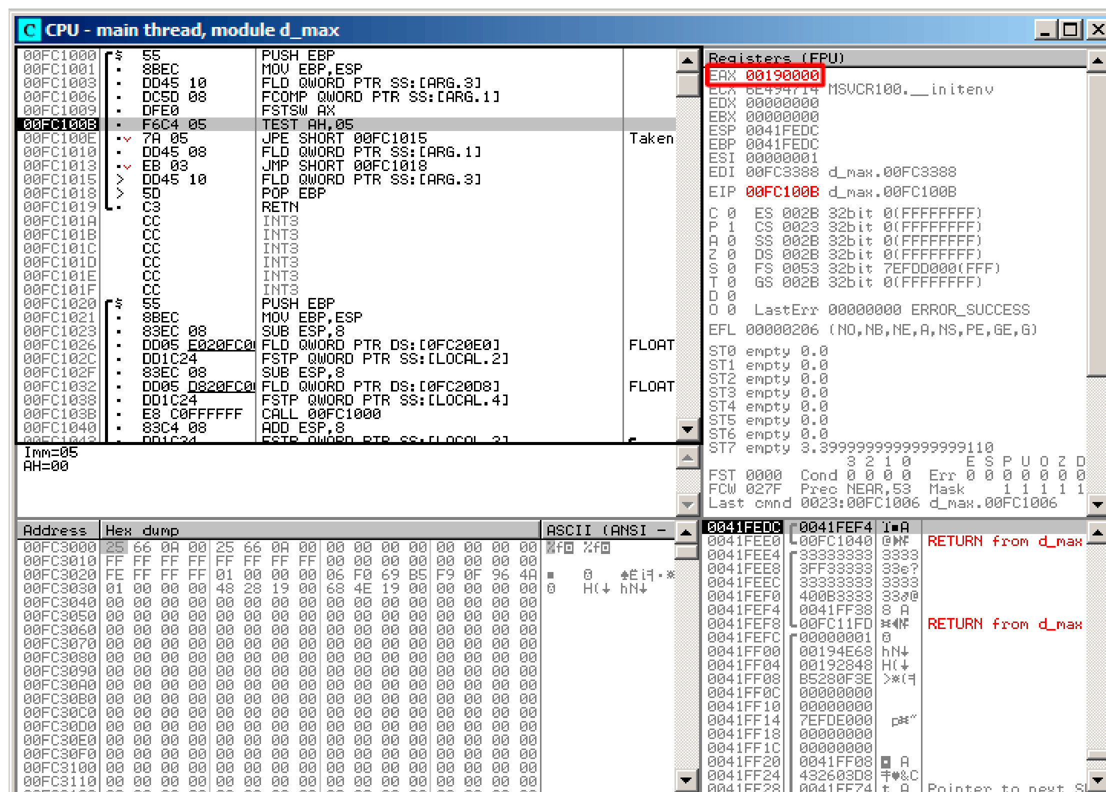


Fig. 1.69: OllyDbg : FNSTSW a été exécutée

Nous voyons que le registre AX contient des zéro: en effet, tous les flags de condition sont à zéro. (OllyDbg désassemble l'instruction FNSTSW comme FSTSW—elles sont synonymes).

TEST a été exécutée:

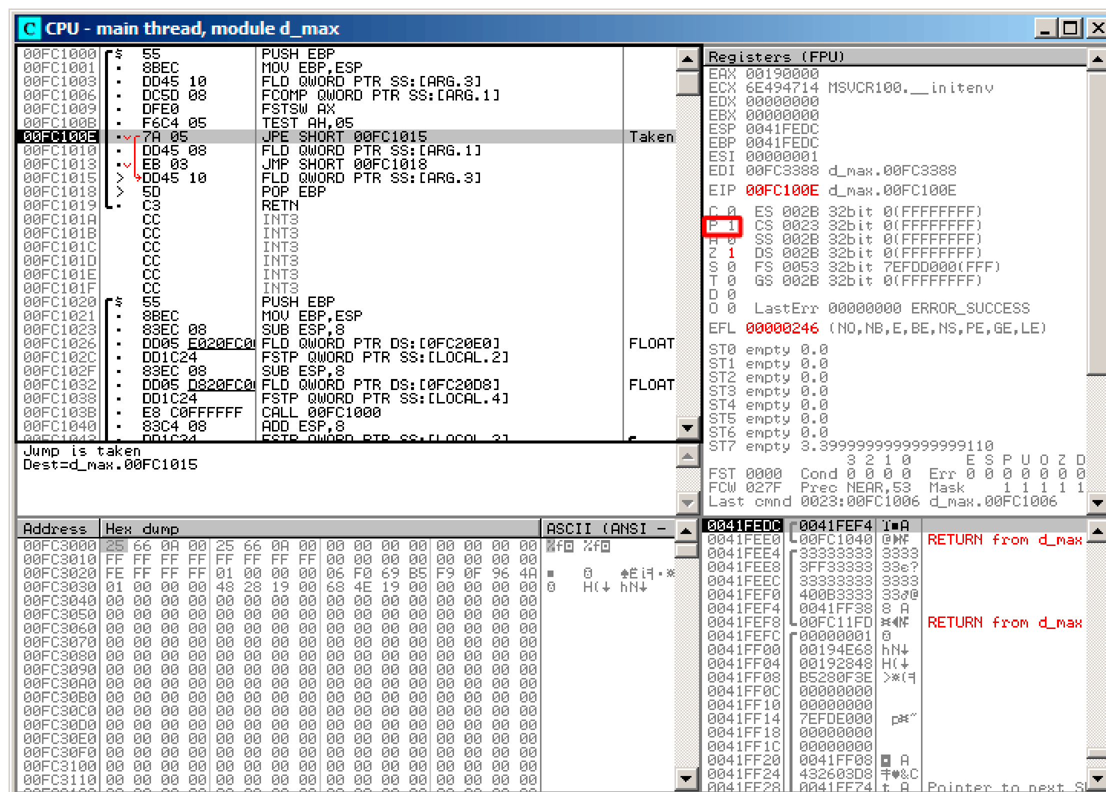


Fig. 1.70: OllyDbg : TEST a été exécutée

Le flag PF est mis à 1.

En effet: le nombre de bit mis à 0 est 0 et 0 est un nombre pair. olly désassemble l'instruction JP comme JPE[115]—elles sont synonymes. Et elle va maintenant se déclencher.

[115]Jump Parity Even (instruction x86)

JPE déclenchée, FLD charge la valeur de b (3.4) dans ST(0) :

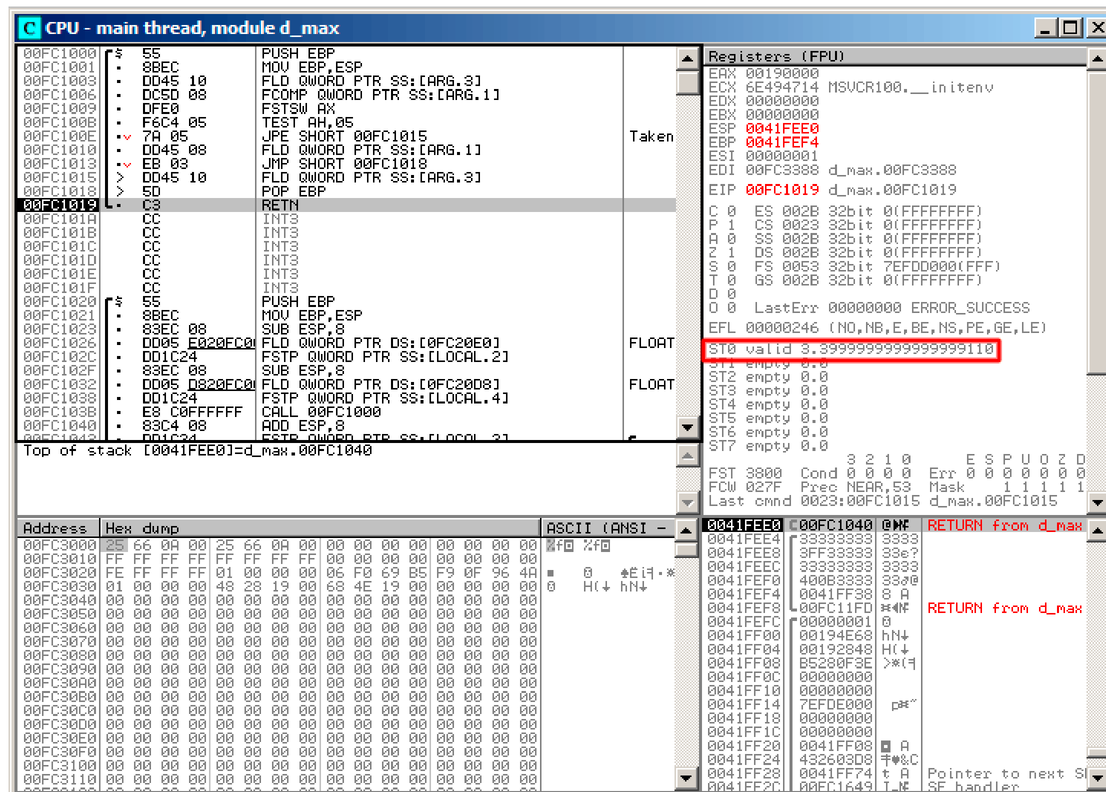


Fig. 1.71: OllyDbg : la seconde instruction FLD a été exécutée

La fonction a fini son travail.

Second exemple sous OllyDbg : a=5.6 et b=-4

Chargeons l'exemple dans OllyDbg :

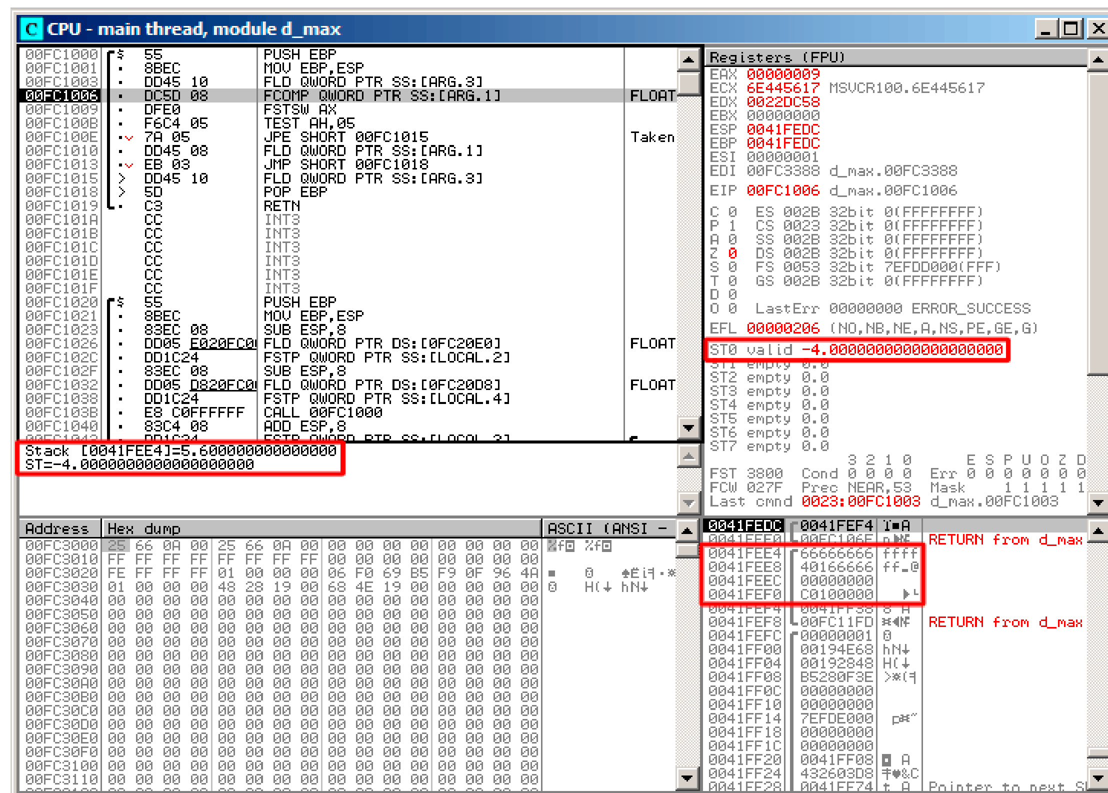


Fig. 1.72: OllyDbg : premier FLD exécutée

Arguments de la fonction courante: $a = 5.6$ et $b = -4$. b (-4) est déjà chargé dans ST(0). FCOMP va s'exécuter maintenant. OllyDbg montre le second argument de FCOMP, qui est sur la pile juste maintenant.

FCOMP a été exécutée:

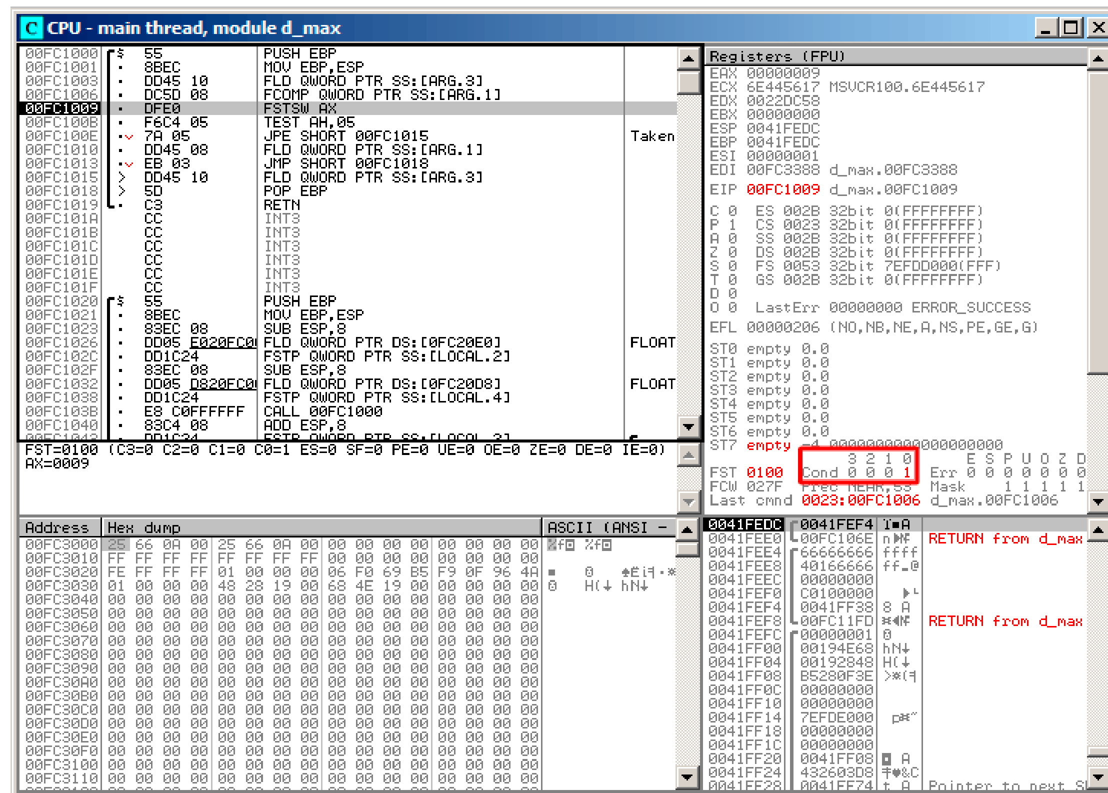


Fig. 1.73: OllyDbg : FCOMP exécutée

Nous voyons l'état des flags de condition du FPU : tous à zéro sauf C0.

FNSTSW a été exécutée:

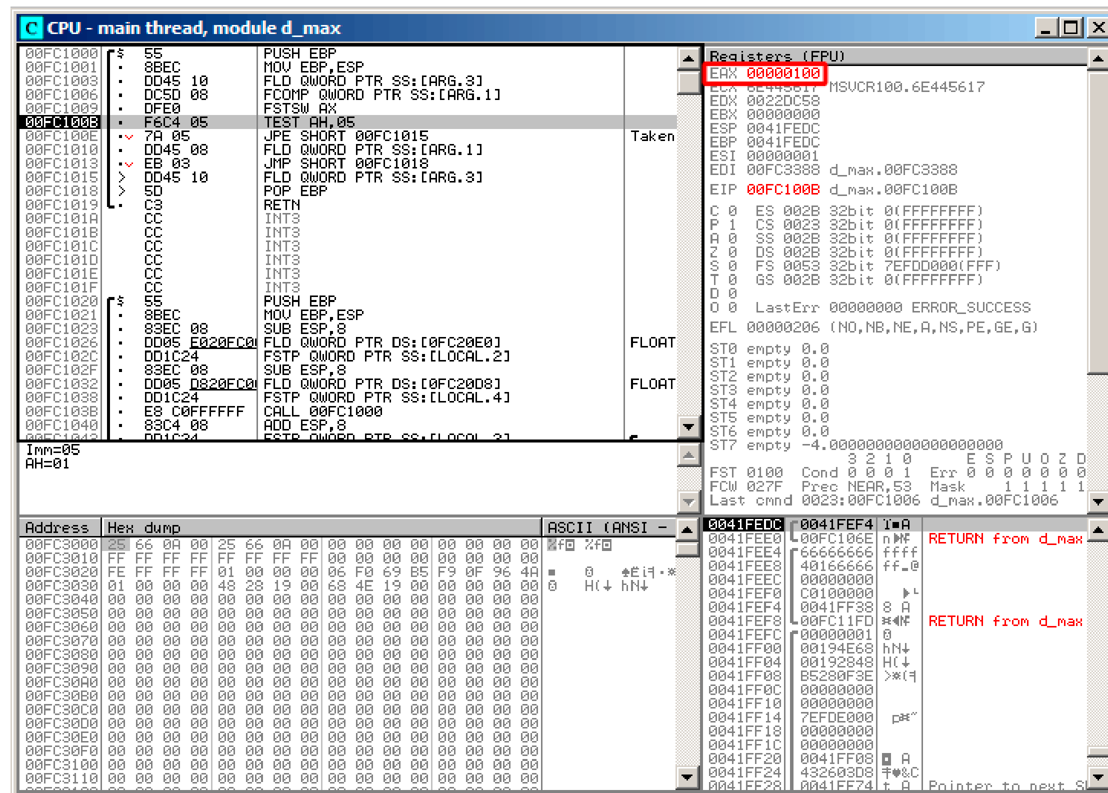


Fig. 1.74: OllyDbg : FNSTSW exécutée

Nous voyons que le registre AX contient `0x100` : le flag `C0` est au 8ième bit.

TEST a été exécutée:

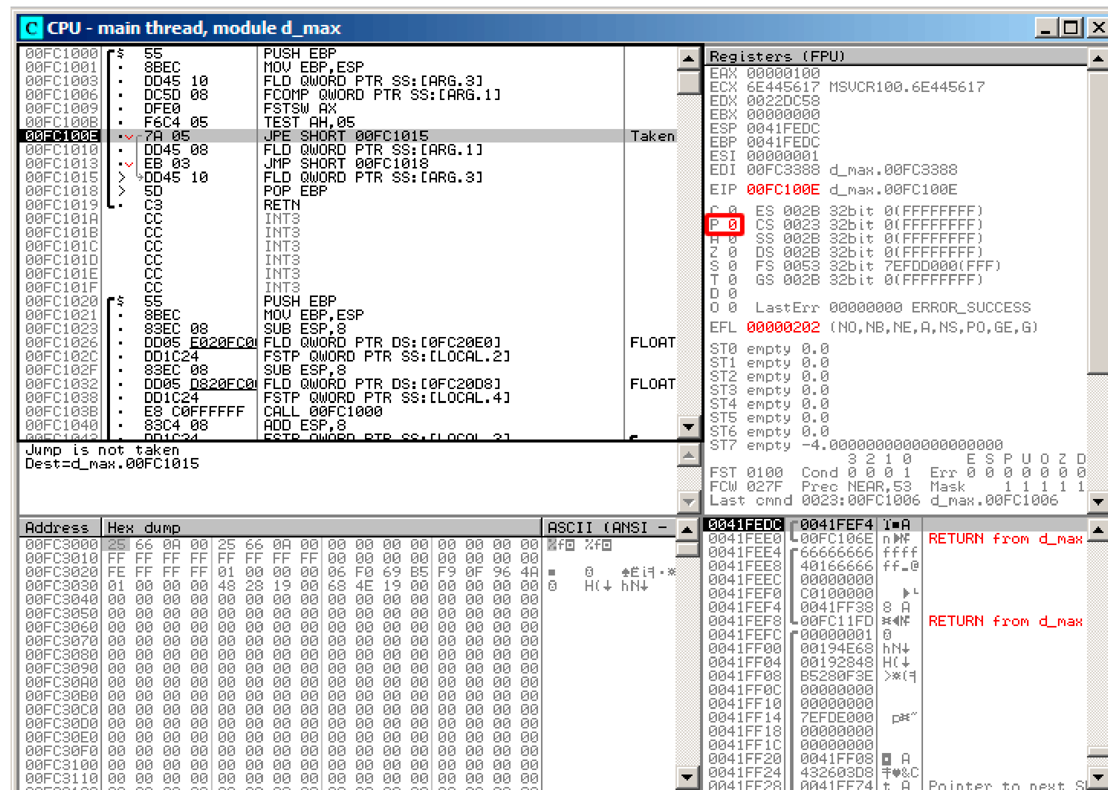


Fig. 1.75: OllyDbg : TEST exécutée

Le flag PF est mis à zéro. En effet:

le nombre de bit mis à 1 dans `0x100` est 1, et 1 est un nombre impair. JPE est sautée maintenant.

JPE n'a pas été déclenchée, donc FLD charge la valeur de a (5.6) dans ST(0) :

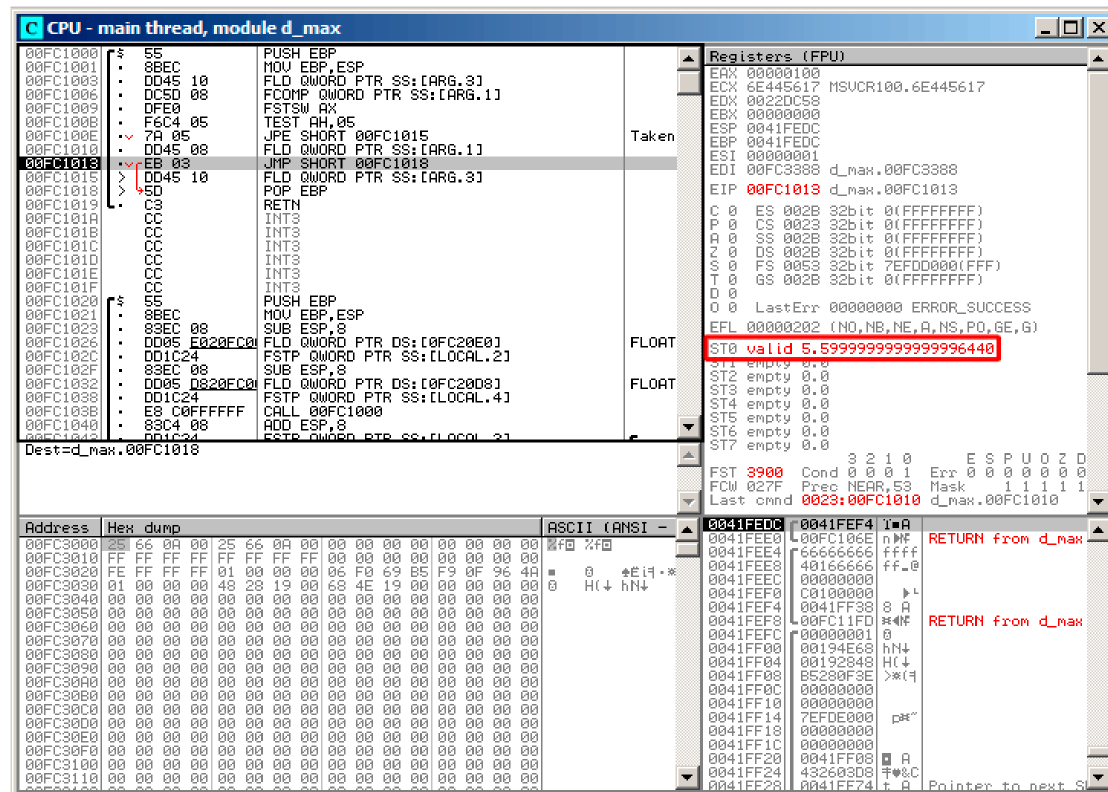


Fig. 1.76: OllyDbg : second FLD exécutée

La fonction a fini son travail.

MSVC 2010 avec optimisation

Listing 1.214 : MSVC 2010 avec optimisation

```
_a$ = 8                  ; size = 8
_b$ = 16                 ; size = 8
_d_max    PROC
    fld     QWORD PTR _b$[esp-4]
    fld     QWORD PTR _a$[esp-4]

; état courant de la pile: ST(0) = _a, ST(1) = _b

    fcom    ST(1) ; comparer _a et ST(1) = (_b)
    fnstsw  ax
    test    ah, 65 ; 00000041H
    jne     SHORT $LN5@d_max
```

```
; copier ST(0) dans ST(1) et dépiler le registre,
; laisser (_a) au sommet
    fstp    ST(1)

; état courant de la pile: ST(0) = _a

    ret     0
$LN5@d_max:
; copier ST(0) dans ST(0) et dépiler le registre,
; laisser (_b) au sommet
    fstp    ST(0)

; état courant de la pile: ST(0) = _b

    ret     0
_d_max    ENDP
```

FCOM diffère de FCOMP dans le sens où il compare seulement les deux valeurs, et ne change pas la pile du FPU. Contrairement à l'exemple précédent, ici les opérandes sont dans l'ordre inverse, c'est pourquoi le résultat de la comparaison dans C3/C2/C0 est différent.

- si $a > b$ dans notre exemple, alors les bits C3/C2/C0 sont mis comme suit: 0, 0, 0.
- si $b > a$, alors les bits sont: 0, 0, 1.
- si $a = b$, alors les bits sont: 1, 0, 0.

L'instruction test ah, 65 laisse seulement deux bits —C3 et C0. Les deux seront à zéro si $a > b$: dans ce cas le saut JNE ne sera pas effectué. Puis FSTP ST(1) suit — cette instruction copie la valeur de ST(0) dans l'opérande et supprime une valeur de la pile du FPU. En d'autres mots, l'instruction copie ST(0) (où la valeur de _a se trouve) dans ST(1). Après cela, deux copies de _a sont sur le sommet de la pile. Puis, une valeur est supprimée. Après cela, ST(0) contient _a et la fonction se termine.

Le saut conditionnel JNE est effectué dans deux cas: si $b > a$ ou $a = b$. ST(0) est copié dans ST(0), c'est comme une opération sans effet (NOP), puis une valeur est supprimée de la pile et le sommet de la pile (ST(0)) contient la valeur qui était avant dans ST(1) (qui est _b). Puis la fonction se termine. La raison pour laquelle cette instruction est utilisée ici est sans doute que le FPU n'a pas d'autre instruction pour prendre une valeur sur la pile et la supprimer.

Premier exemple sous OllyDbg : a=1.2 et b=3.4

Les deux instructions FLD ont été exécutées:

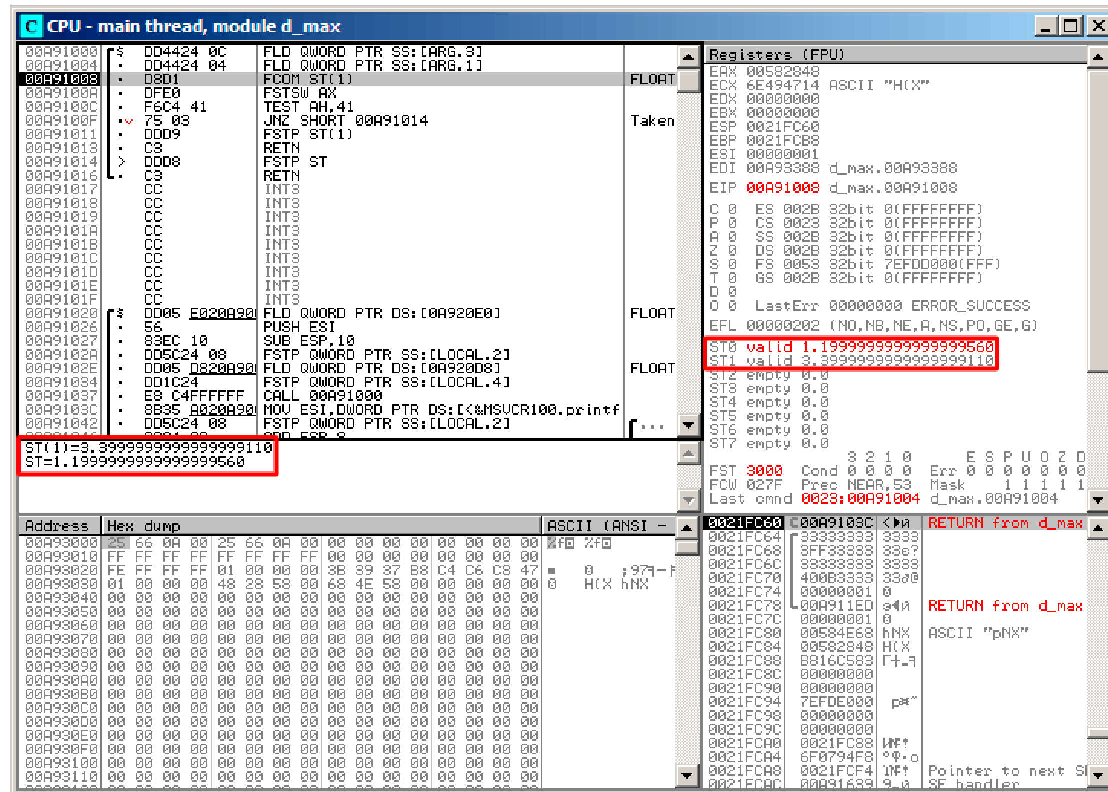


Fig. 1.77: OllyDbg : les deux FLD exécutées

FCOM exécutée: OllyDbg montre le contenu de ST(0) et ST(1) par commodité.

FCOM a été exécutée:

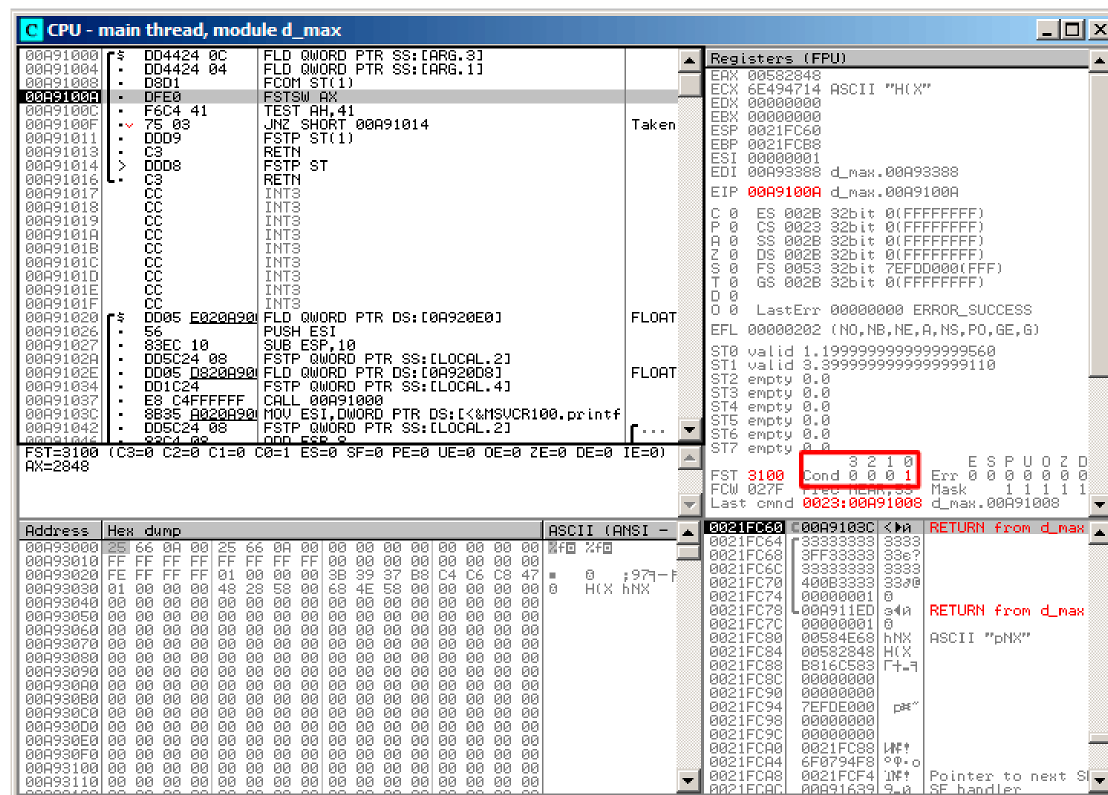


Fig. 1.78: OllyDbg : FCOM a été exécutée

C0 est mis, tous les autres flags de condition sont à zéro.

FNSTSW a été exécutée, AX=0x3100:

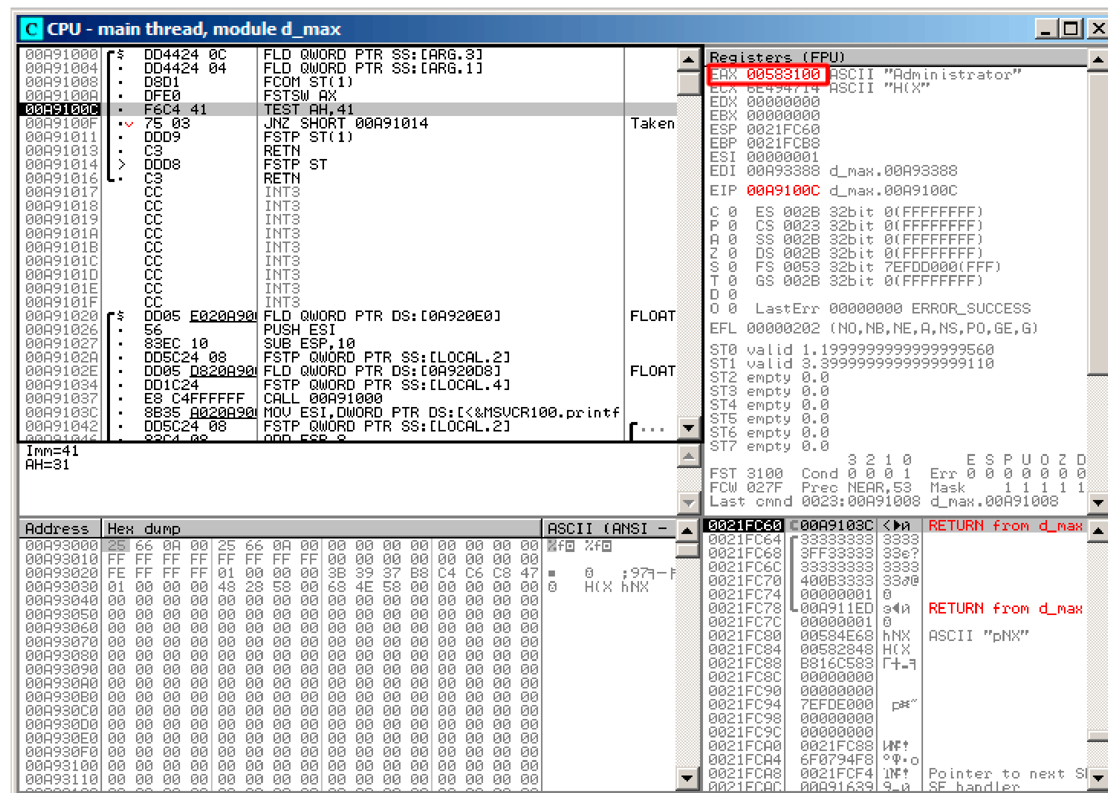


Fig. 1.79: OllyDbg : FNSTSW est exécutée

TEST est exécutée:

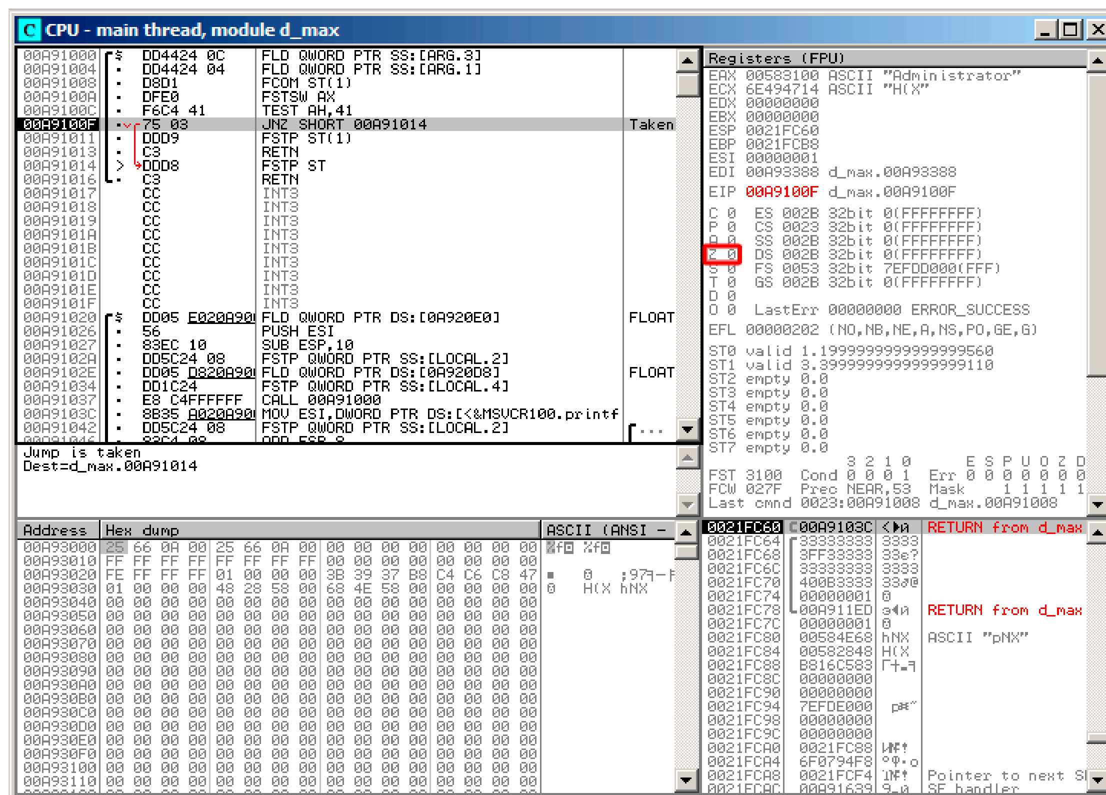


Fig. 1.80: OllyDbg : TEST est exécutée

ZF=0, le saut conditionnel va être déclenché maintenant.

FSTP ST (ou FSTP ST(0)) a été exécuté —1.2 a été dépilé, et 3.4 laissé au sommet de la pile:

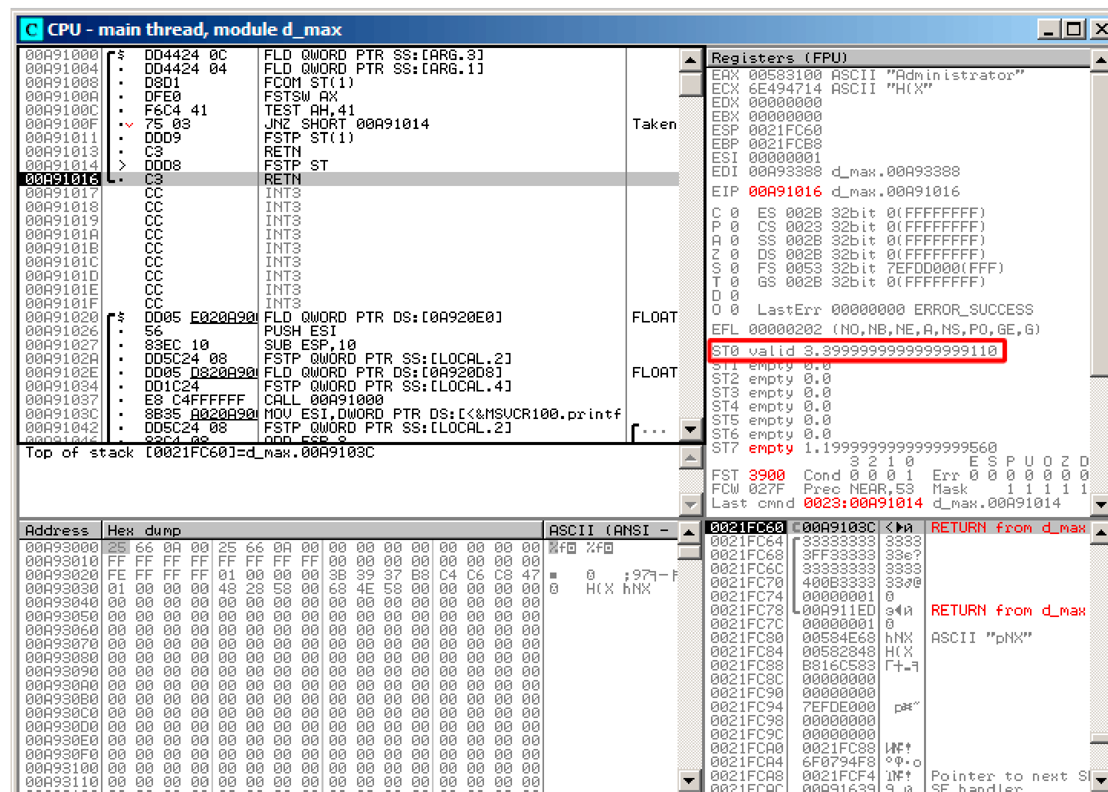


Fig. 1.81: OllyDbg : FSTP est exécutée

Nous voyons que l’instruction FSTP ST

fonctionne comme dépiler une valeur de la pile du FPU.

Second exemple sous OllyDbg : a=5.6 et b=-4

Les deux FLD sont exécutées:

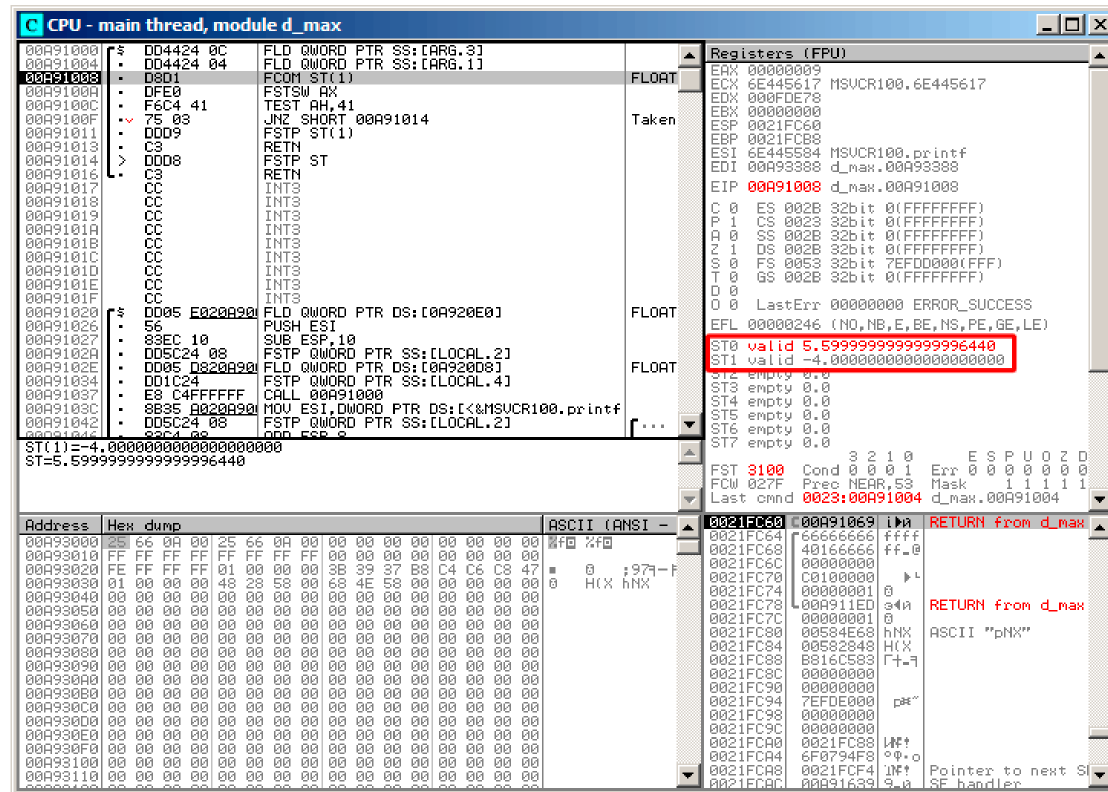


Fig. 1.82: OllyDbg : les deux FLD sont exécutée

FCOM est sur le point de s'exécuter.

FCOM a été exécutée:

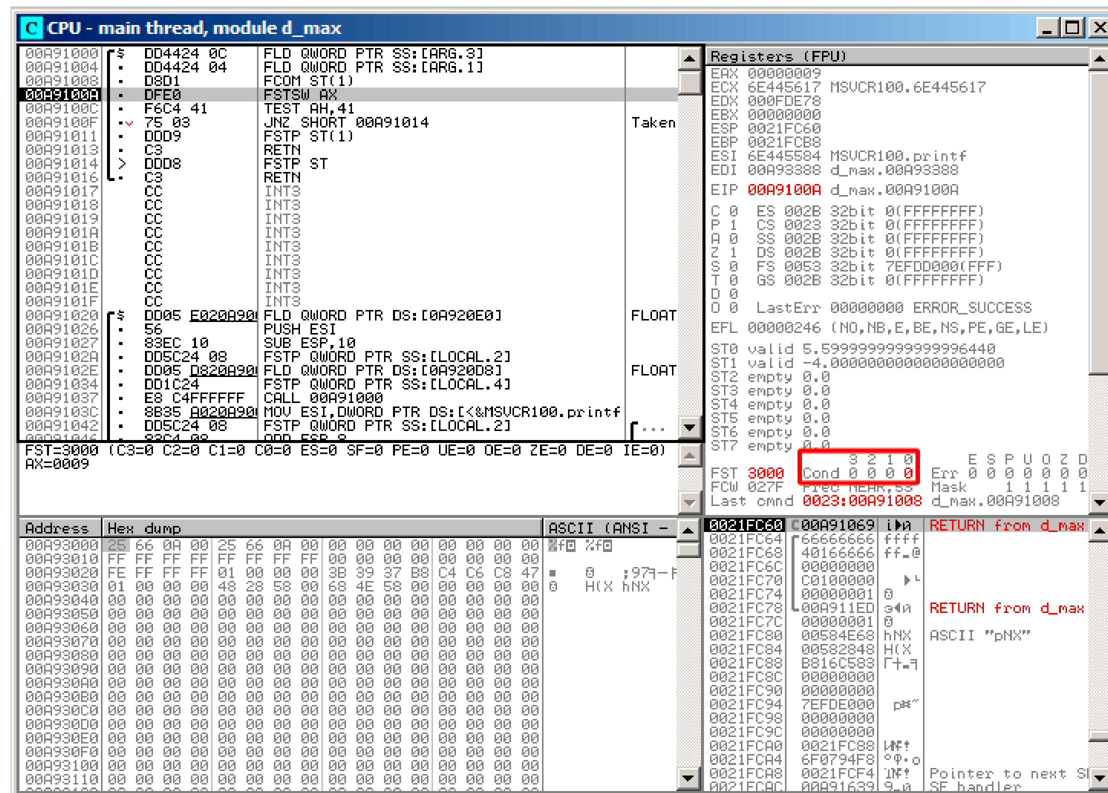


Fig. 1.83: OllyDbg : FCOM est terminé

Tous les flags de conditions sont à zéro.

FNSTSW fait, AX=0x3000:

Fig. 1.84: OllyDbg : FNSTSW a été exécutée

TEST a été exécutée:

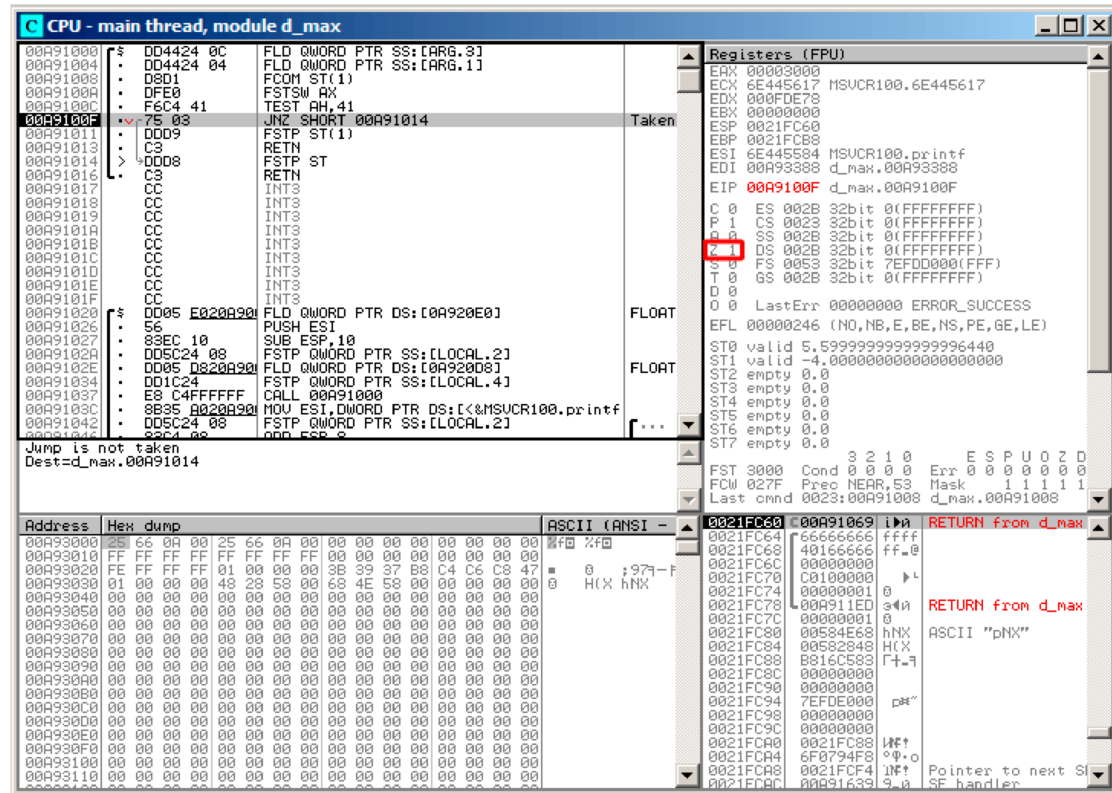


Fig. 1.85: OllyDbg : TEST a été exécutée

ZF=1, le saut ne va pas se produire maintenant.

FSTP ST(1) a été exécutée: une valeur de 5.6 est maintenant au sommet de la pile du FPU.

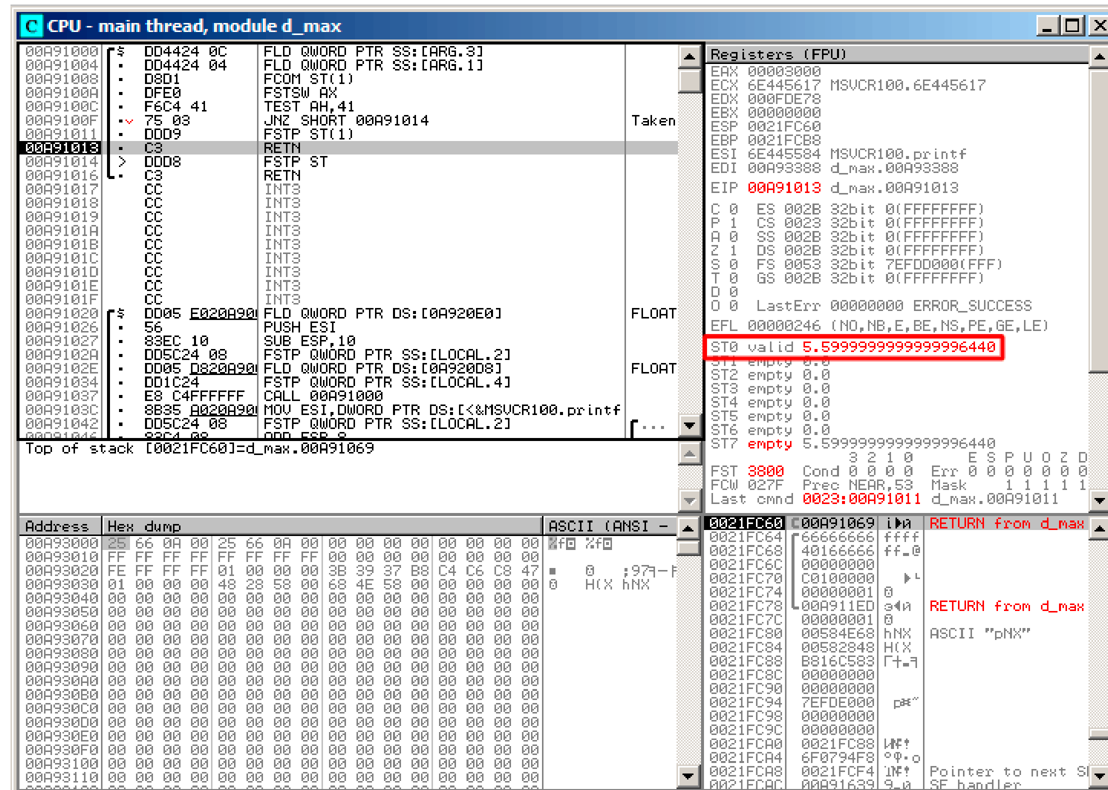


Fig. 1.86: OllyDbg : FSTP a été exécutée

Nous voyons maintenant que l'instruction FSTP ST(1) fonctionne comme suit: elle laisse ce qui était au sommet de la pile, mais met ST(1) à zéro.

GCC 4.4.1

Listing 1.215 : GCC 4.4.1

```
d_max proc near

b               = qword ptr -10h
a               = qword ptr -8
a_first_half    = dword ptr  8
a_second_half   = dword ptr  0Ch
b_first_half    = dword ptr  10h
b_second_half   = dword ptr  14h

    push    ebp
```

```
    mov     ebp, esp
    sub     esp, 10h

; mettre a et b sur la pile locale:

    mov     eax, [ebp+a_first_half]
    mov     dword ptr [ebp+a], eax
    mov     eax, [ebp+a_second_half]
    mov     dword ptr [ebp+a+4], eax
    mov     eax, [ebp+b_first_half]
    mov     dword ptr [ebp+b], eax
    mov     eax, [ebp+b_second_half]
    mov     dword ptr [ebp+b+4], eax

; charger a et b sur la pile du FPU:

    fld     [ebp+a]
    fld     [ebp+b]

; état courant de la pile: ST(0) - b; ST(1) - a

    fxch    st(1) ; cette instruction échange ST(1) et ST(0)

; état courant de la pile: ST(0) - a; ST(1) - b

    fucompp    ; comparer a et b et prendre deux valeurs depuis la pile,
    i.e., a et b
    fnstsw  ax ; stocker l'état du FPU dans AX
    sahf       ; charger l'état des flags SF, ZF, AF, PF, et CF depuis AH
    setnbe  al ; mettre 1 dans AL, si CF=0 et ZF=0
    test    al, al            ; AL==0 ?
    jz      short loc_8048453 ; oui
    fld     [ebp+a]
    jmp     short locret_8048456

loc_8048453:
    fld     [ebp+b]

locret_8048456:
    leave
    retn
d_max endp
```

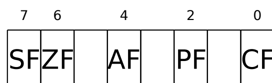
FUCOMPP est presque comme FCOM, mais dépile deux valeurs de la pile et traite les «non-nombres» différemment.

Quelques informations à propos des *not-a-numbers* (non-nombres).

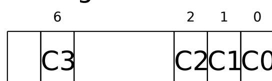
Le FPU est capable de traiter les valeurs spéciales que sont les *not-a-numbers* (non-nombres) ou NaNs. Ce sont les infinis, les résultat de division par 0, etc. Les non-nombres peuvent être «quiet» et «signaling». Il est possible de continuer à travailler avec les «quiet» NaNs, mais si l'on essaye de faire une opération avec un «signaling» NaNs, une exception est levée.

FCOM lève une exception si un des opérandes est NaN. FUCOM lève une exception seulement si un des opérandes est un signaling NaN (SNaN).

L'instruction suivante est SAHF (*Store AH into Flags* stocker AH dans les Flags) —est une instruction rare dans le code non relatif au FPU. 8 bits de AH sont copiés dans les 8-bits bas dans les flags du CPU dans l'ordre suivant:



Rappelons que FNSTSW déplace des bits qui nous intéressent (C3/C2/C0) dans AH et qu'ils sont aux positions 6, 2, 0 du registre AH.



En d'autres mots, la paire d'instructions `fnstsw ax / sahf` déplace C3/C2/C0 dans ZF, PF et CF.

Maintenant, rappelons les valeurs de C3/C2/C0 sous différentes conditions:

- Si a est plus grand que b dans notre exemple, alors les C3/C2/C0 sont mis à: 0, 0, 0.
- Si a est plus petit que b, alors les bits sont mis à: 0, 0, 1.
- Si $a = b$, alors: 1, 0, 0.

En d'autres mots, ces états des flags du CPU sont possible après les trois instructions FUCOMPP/FNSTSW/SAHF :

- Si $a > b$, les flags du CPU sont mis à: `ZF=0, PF=0, CF=0`.
- Si $a < b$, alors les flags sont mis à: `ZF=0, PF=0, CF=1`.
- Et si $a = b$, alors: `ZF=1, PF=0, CF=0`.

Suivant les flags du CPU et les conditions, SETNBE met 1 ou 0 dans AL. C'est presque la contrepartie de JNBE, avec l'exception que SETcc[116] met 1 ou 0 dans AL, mais Jcc effectue un saut ou non. SETNBE met 1 seulement si CF=0 et ZF=0. Si ce n'est pas vrai, 0 est mis dans AL.

Il y a un seul cas où CF et ZF sont à 0: si $a > b$.

Alors 1 est mis dans AL, le JZ subséquent n'est pas pris et la fonction va renvoyer _a. Dans tous les autres cas, _b est renvoyé.

GCC 4.4.1 avec optimisation

Listing 1.216 : GCC 4.4.1 avec optimisation

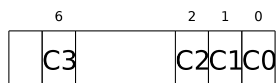
```
                public d_max
d_max           proc near
```

[116] *cc* est un *condition code*

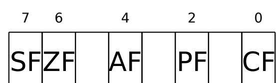
```
arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                mov     ebp, esp
                fld     [ebp+arg_0] ; _a
                fld     [ebp+arg_8] ; _b

; état de la pile maintenant: ST(0) = _b, ST(1) = _a
                fxch    st(1)

; état de la pile maintenant: ST(0) = _a, ST(1) = _b
                fucom   st(1) ; comparer _a et _b
                fnstsw  ax
                sahf
                ja      short loc_8048448

; stocker ST(0) dans ST(0) (opération sans effet),
; dépiler une valeur du sommet de la pile,
; laisser _b au sommet
                fstp    st
                jmp     short loc_804844A

loc_8048448:
; stocker _a dans ST(1), dépiler une valeur du sommet de la pile, laisser _a
    au sommet
                fstp    st(1)

loc_804844A:
                pop     ebp
                retn
d_max           endp
```

C'est presque le même, à l'exception que JA est utilisé après SAHF. En fait, les instructions de sauts conditionnels qui vérifient «plus», «moins» ou «égal» pour les comparaisons de nombres non signés (ce sont JA, JAE, JB, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ) vérifient seulement les flags CF et ZF.

Rappelons comment les bits C3/C2/C0 sont situés dans le registre AH après l'exécution de FSTSW/FNSTSW :



Rappelons également, comment les bits de AH sont stockés dans les flags du CPU après l'exécution de SAHF :



Après la comparaison, les bits C3 et C0 sont copiés dans ZF et CF, donc les sauts conditionnels peuvent fonctionner après. st déclenché si CF et ZF sont tout les deux à zéro.

Ainsi, les instructions de saut conditionnel listées ici peuvent être utilisées après une paire d'instructions FNSTSW/SAHF.

Apparemment, les bits d'état du FPU C3/C2/C0 ont été mis ici intentionnellement, pour facilement les relier aux flags du CPU de base sans permutations supplémentaires?

GCC 4.8.1 avec l'option d'optimisation -O3

De nouvelles instructions FPU ont été ajoutées avec la famille Intel P6[117]. Ce sont FUCOMI (comparer les opérandes et positionner les flags du CPU principal) et FCMOVcc (fonctionne comme CMOVcc, mais avec les registres du FPU).

Apparemment, les mainteneurs de GCC ont décidé de supprimer le support des CPUs Intel pré-P6 (premier Pentium, 80486, etc.).

Et donc, le FPU n'est plus une unité séparée dans la famille Intel P6, ainsi il est possible de modifier/vérifier un flag du CPU principal depuis le FPU.

Voici ce que nous obtenons:

Listing 1.217 : GCC 4.8.1 avec optimisation

```
        fld     QWORD PTR [esp+4]       ; charger "a"
        fld     QWORD PTR [esp+12]      ; charger "b"
        ; ST0=b, ST1=a
        fxch    st(1)
        ; ST0=a, ST1=b
        ; comparer "a" et "b"
        fucomi  st, st(1)
        ; copier ST1 ("b" ici) dans ST0 si a<=b
        ; laisser "a" dans ST0 autrement
        fcmovbe st, st(1)
        ; supprimer la valeur dans ST1
        fstp    st(1)
        ret
```

Difficile de deviner pourquoi FXCH (échange les opérandes) est ici.

Il est possible de s'en débarrasser facilement en échangeant les deux premières instructions FLD ou en remplaçant FCMOVBE (*below or equal* inférieur ou égal) par FCMOVA (*above*). Il s'agit probablement d'une imprécision du compilateur.

Donc FUCOMI compare ST(0) (a) et ST(1) (b) et met certains flags dans le CPU principal. FCMOVBE vérifie les flags et copie ST(1) (b ici à ce moment) dans ST(0) (a ici) si $ST0(a) <= ST1(b)$. Autrement ($a > b$), a est laissé dans ST(0).

Le dernier FSTP laisse ST(0) sur le sommet de la pile, supprimant le contenu de ST(1).

Exécutons pas à pas cette fonction dans GDB:

[117] À partir du Pentium Pro, Pentium-II, etc.

Listing 1.218 : GCC 4.8.1 avec optimisation and GDB

```
dennis@ubuntuvm:~/polygon$ gcc -O3 d_max.c -o d_max -fno-inline
dennis@ubuntuvm:~/polygon$ gdb d_max
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/d_max...(no debugging symbols found)...done.
(gdb) b d_max
Breakpoint 1 at 0x80484a0
(gdb) run
Starting program: /home/dennis/polygon/d_max

Breakpoint 1, 0x080484a0 in d_max ()
(gdb) ni
0x080484a4 in d_max ()
(gdb) disas $eip
Dump of assembler code for function d_max:
   0x080484a0 <+0>:     fldl   0x4(%esp)
=> 0x080484a4 <+4>:     fldl   0xc(%esp)
   0x080484a8 <+8>:     fxch   %st(1)
   0x080484aa <+10>:    fucomi %st(1),%st
   0x080484ac <+12>:    fcmovbe %st(1),%st
   0x080484ae <+14>:    fstp   %st(1)
   0x080484b0 <+16>:    ret
End of assembler dump.
(gdb) ni
0x080484a8 in d_max ()
(gdb) info float
  R7: Valid   0x3fff9999999999999800 +1.199999999999999956
=>R6: Valid   0x4000d999999999999800 +3.399999999999999911
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
  R0: Empty   0x00000000000000000000

Status Word:         0x3000
                       TOP: 6
Control Word:        0x037f   IM DM ZM OM UM PM
                       PC: Extended Precision (64-bits)
                       RC: Round to nearest
Tag Word:            0x0fff
Instruction Pointer: 0x73:0x080484a4
Operand Pointer:     0x7b:0xbffff118
Opcode:              0x0000
(gdb) ni
0x080484aa in d_max ()
(gdb) info float
  R7: Valid   0x4000d999999999999800 +3.399999999999999911
=>R6: Valid   0x3fff9999999999999800 +1.199999999999999956
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
```

```
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
  R0: Empty   0x00000000000000000000

Status Word:         0x3000
                       TOP: 6
Control Word:        0x037f   IM DM ZM OM UM PM
                       PC: Extended Precision (64-bits)
                       RC: Round to nearest
Tag Word:            0x0fff
Instruction Pointer: 0x73:0x080484a8
Operand Pointer:     0x7b:0xbffff118
Opcode:              0x0000
(gdb) disas $eip
Dump of assembler code for function d_max:
   0x080484a0 <+0>:     fldl   0x4(%esp)
   0x080484a4 <+4>:     fldl   0xc(%esp)
   0x080484a8 <+8>:     fxch   %st(1)
=> 0x080484aa <+10>:    fucomi %st(1),%st
   0x080484ac <+12>:    fcmovbe %st(1),%st
   0x080484ae <+14>:    fstp   %st(1)
   0x080484b0 <+16>:    ret
End of assembler dump.
(gdb) ni
0x080484ac in d_max ()
(gdb) info registers
eax            0x1      1
ecx            0xbffff1c4       -1073745468
edx            0x8048340        134513472
ebx            0xb7fbf000       -1208225792
esp            0xbffff10c       0xbffff10c
ebp            0xbffff128       0xbffff128
esi            0x0      0
edi            0x0      0
eip            0x80484ac        0x80484ac <d_max+12>
eflags         0x203    [ CF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) ni
0x080484ae in d_max ()
(gdb) info float
  R7: Valid   0x4000d999999999999800 +3.399999999999999911
=>R6: Valid   0x4000d999999999999800 +3.399999999999999911
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
```

```
105   R0: Empty   0x00000000000000000000
106
107 Status Word:         0x3000
108                        TOP: 6
109 Control Word:        0x037f   IM DM ZM OM UM PM
110                        PC: Extended Precision (64-bits)
111                        RC: Round to nearest
112 Tag Word:            0x0fff
113 Instruction Pointer: 0x73:0x080484ac
114 Operand Pointer:     0x7b:0xbffff118
115 Opcode:              0x0000
116 (gdb) disas $eip
117 Dump of assembler code for function d_max:
118    0x080484a0 <+0>:     fldl   0x4(%esp)
119    0x080484a4 <+4>:     fldl   0xc(%esp)
120    0x080484a8 <+8>:     fxch   %st(1)
121    0x080484aa <+10>:    fucomi %st(1),%st
122    0x080484ac <+12>:    fcmovbe %st(1),%st
123 => 0x080484ae <+14>:    fstp   %st(1)
124    0x080484b0 <+16>:    ret
125 End of assembler dump.
126 (gdb) ni
127 0x080484b0 in d_max ()
128 (gdb) info float
129 =>R7: Valid   0x4000d999999999999800 +3.399999999999999911
130   R6: Empty   0x4000d999999999999800
131   R5: Empty   0x00000000000000000000
132   R4: Empty   0x00000000000000000000
133   R3: Empty   0x00000000000000000000
134   R2: Empty   0x00000000000000000000
135   R1: Empty   0x00000000000000000000
136   R0: Empty   0x00000000000000000000
137
138 Status Word:         0x3800
139                        TOP: 7
140 Control Word:        0x037f   IM DM ZM OM UM PM
141                        PC: Extended Precision (64-bits)
142                        RC: Round to nearest
143 Tag Word:            0x3fff
144 Instruction Pointer: 0x73:0x080484ae
145 Operand Pointer:     0x7b:0xbffff118
146 Opcode:              0x0000
147 (gdb) quit
148 A debugging session is active.
149
150         Inferior 1 [process 30194] will be killed.
151
152 Quit anyway? (y or n) y
153 dennis@ubuntuvm:~/polygon$
```

En utilisant «ni », exécutons les deux premières instructions FLD.

Examinons les registres du FPU (ligne 33).

Comme cela a déjà été mentionné, l'ensemble des registres FPU est un buffeur circulaire plutôt qu'une pile (1.25.5 on page 293). Et GDB ne montre pas les registres STx, mais les registre internes du FPU (Rx). La flèche (à la ligne 35) pointe sur le haut courant de la pile.

Vous pouvez voir le contenu du registre TOP dans le *Status Word* (ligne 36-37)—c'est 6 maintenant, donc le haut de la pile pointe maintenant sur le registre interne 6.

Les valeurs de a et b sont échangées après l'exécution de FXCH (ligne 54).

FUCOMI est exécuté (ilgne 83). Regardons les flags: CF est mis (ligne 95).

FCMOVBE a copié la valeur de b (voir ligne 104).

FSTP dépose une valeur au sommet de la pile (ligne 139). La valeur de TOP est maintenant 7, donc le sommet de la pile du FPU pointe sur le registre interne 7.

ARM

avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Listing 1.219 : avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
VMOVGT.F64      D16, D17 ; copier "a" dans D16
VMOV            R0, R1, D16
BX              LR
```

Un cas très simple. Les valeurs en entrée sont placées dans les registres D17 et D16 puis comparées en utilisant l'instruction VCMPE.

Tout comme dans le coprocesseur x86, le coprocesseur ARM a son propre registre de flags (FPSCR[118]), puisqu'il est nécessaire de stocker des flags spécifique au coprocesseur. Et tout comme en x86, il n'y a pas d'instruction de saut conditionnel qui teste des bits dans le registre de status du coprocesseur. Donc il y a VMRS, qui copie 4 bits (N, Z, C, V) du mot d'état du coprocesseur dans les bits du registre de status *général* (APSR[119]).

VMOVGT est l'analogue de l'instruction MOVGT pour D-registres, elle s'exécute si un opérande est plus grand que l'autre lors de la comparaison (*GT—Greater Than*).

Si elle est exécutée, la valeur de a sera écrite dans D16 (ce qui est écrit en ce moment dans D17). Sinon, la valeur de b reste dans le registre D16.

La pénultième instruction VMOV prépare la valeur dans la registre D16 afin de la renvoyer dans la paire de registres R0 et R1.

[118](ARM) Floating-Point Status and Control Register
[119](ARM) Application Program Status Register

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

Listing 1.220 : avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
IT GT
VMOVGT.F64      D16, D17
VMOV            R0, R1, D16
BX              LR
```

Presque comme dans l'exemple précédent, toutefois légèrement différent. Comme nous le savons déjà, en mode ARM, beaucoup d'instructions peuvent avoir un prédicat de condition. Mais il n'y a rien de tel en mode Thumb. Il n'y a pas d'espace dans les instructions sur 16-bit pour 4 bits dans lesquels serait encodée la condition.

Toutefois, cela à été étendu en un mode Thumb-2 pour rendre possible de spécifier un prédicat aux instructions de l'ancien mode Thumb. Ici, dans le listing généré par IDA, nous voyons l'instruction `VMOVGT`, comme dans l'exemple précédent.

En fait, le `VMOV` usuel est encodé ici, mais IDA lui ajoute le suffixe `-GT`, puisque que l'instruction `IT GT` se trouve juste avant.

L'instruction `IT` défini ce que l'on appelle un *bloc if-then*.

Après cette instruction, il est possible de mettre jusqu'à 4 instructions, chacune d'entre elles ayant un suffixe de prédicat. Dans notre exemple, `IT GT` implique que l'instruction suivante ne sera exécutée que si la condition *GT* (*Greater Than* plus grand que) est vraie.

Voici un exemple de code plus complexe, à propos, d'Angry Birds (pour iOS) :

Listing 1.221 : Angry Birds Classic

```
...
ITE NE
VMOVNE          R2, R3, D16
VMOVEQ          R2, R3, D17
BLX             _objc_msgSend ; not suffixed
...
```

`ITE` est l'acronyme de *if-then-else* et elle encode un suffixe pour les deux prochaines instructions.

La première instruction est exécutée si la condition encodée dans `ITE` (*NE, not equal*) est vraie, et la seconde—si la condition n'est pas vraie (l'inverse de la condition `NE` est `EQ` (*equal*)).

L'instruction qui suit le second `VMOV` (ou `VMOVEQ`) est normale, non suffixée (`BLX`).

Un autre exemple qui est légèrement plus difficile, qui est aussi d'Angry Birds:

Listing 1.222 : Angry Birds Classic

```
...
ITTTT EQ
MOVEQ           R0, R4
ADDEQ           SP, SP, #0x20
POPEQ.W         {R8,R10}
POPEQ           {R4-R7,PC}
BLX             ___stack_chk_fail ; not suffixed
...
```

Les quatre symboles «T» dans le mnémonique de l'instruction signifient que les quatre instructions suivantes seront exécutées si la condition est vraie.

C'est pourquoi IDA ajoute le suffixe -EQ à chacune d'entre elles.

Et si il y avait, par exemple, ITEEE EQ (*if-then-else-else-else*), alors les suffixes seraient mis comme suit:

```
-EQ
-NE
-NE
-NE
```

Un autre morceau de code d'Angry Birds:

Listing 1.223 : Angry Birds Classic

```
...
CMP.W           R0, #0xFFFFFFFF
ITTE LE
SUBLE.W         R10, R0, #1
NEGLE           R0, R0
MOVGT           R10, R0
MOVS            R6, #0         ; not suffixed
CBZ             R0, loc_1E7E32 ; not suffixed
...
```

ITTE (*if-then-then-else*)

implique que les 1ère et 2ème instructions seront exécutées si la condition LE (*Less or Equal* moins ou égal) est vraie, et que la 3ème—si la condition inverse (GT—*Greater Than* plus grand que) est vraie.

En général, les compilateurs ne génèrent pas toutes les combinaisons possible.

Par exemple, dans le jeu Angry Birds mentionné ((*classic* version pour iOS) seules les les variantes suivantes de l'instruction IT sont utilisées: IT, ITE, ITT, ITTE, ITTT, ITTTT. Comment savoir cela? Dans IDA, il est possible de produire un listing dans un fichier, ce qui a été utilisé pour en créer un avec l'option d'afficher 4 octets pour chaque opcode. Ensuite, en connaissant la partie haute de l'opcode de 16-bit (0xBF pour IT), nous utilisons grep ainsi:

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

À propos, si vous programmez en langage d'assemblage ARM pour le mode Thumb-2, et que vous ajoutez des suffixes conditionnels, l'assembleur ajoutera automatiquement l'instruction IT avec les flags là où ils sont nécessaires.

sans optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Listing 1.224 : sans optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
b               = -0x20
a               = -0x18
val_to_return   = -0x10
saved_R7        = -4

                STR             R7, [SP,#saved_R7]!
                MOV             R7, SP
                SUB             SP, SP, #0x1C
                BIC             SP, SP, #7
                VMOV            D16, R2, R3
                VMOV            D17, R0, R1
                VSTR            D17, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+b]
                VLDR            D16, [SP,#0x20+a]
                VLDR            D17, [SP,#0x20+b]
                VCMPE.F64       D16, D17
                VMRS            APSR_nzcv, FPSCR
                BLE             loc_2E08
                VLDR            D16, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+val_to_return]
                B               loc_2E10

loc_2E08
                VLDR            D16, [SP,#0x20+b]
                VSTR            D16, [SP,#0x20+val_to_return]

loc_2E10
                VLDR            D16, [SP,#0x20+val_to_return]
                VMOV            R0, R1, D16
                MOV             SP, R7
                LDR             R7, [SP+0x20+b],#4
                BX              LR
```

Presque la même chose que nous avons déjà vu, mais ici il y a beaucoup de code redondant car les variables a et b sont stockées sur la pile locale, tout comme la valeur de retour.

avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.225 : avec optimisation Keil 6/2013 (Mode Thumb)

```
                PUSH    {R3-R7,LR}
                MOVS    R4, R2
```

```
                MOVS    R5, R3
                MOVS    R6, R0
                MOVS    R7, R1
                BL      __aeabi_cdrcmple
                BCS     loc_1C0
                MOVS    R0, R6
                MOVS    R1, R7
                POP     {R3-R7,PC}

loc_1C0
                MOVS    R0, R4
                MOVS    R1, R5
                POP     {R3-R7,PC}
```

Keil ne génère pas les instructions pour le FPU car il ne peut pas être sûr qu'elles sont supportées sur le CPU cible, et cela ne peut pas être fait directement en comparant les bits. Donc il appelle une fonction d'une bibliothèque externe pour effectuer la comparaison: `__aeabi_cdrcmple`.

N.B. Le résultat de la comparaison est laissé dans les flags par cette fonction, donc l'instruction BCS (*Carry set—Greater than or equal* plus grand ou égal) fonctionne sans code additionnel.

ARM64

GCC (Linaro) 4.9 avec optimisation

```
d_max:
; D0 - a, D1 - b
        fcmpe   d0, d1
        fcsel   d0, d0, d1, gt
; maintenant le résultat est dans D0
        ret
```

L'ARM64 ISA possède des instructions FPU qui mettent les flags CPU APSR au lieu de FPSCR, par commodité. Le FPU n'est plus un device séparé (au moins, logiquement). Ici, nous voyons FCMPE. Ceci compare les deux valeurs passées dans `D0` et `D1` (qui sont le premier et le second argument de la fonction) et met les flags APSR (N, Z, C, V).

FCSEL (*Floating Conditional Select* (sélection de flottant conditionnelle) copie la valeur de `D0` ou `D1` dans `D0` suivant le résultat de la comparaison (GT—Greater Than), et de nouveau, il utilise les flags dans le registre APSR au lieu de FPSCR.

Ceci est bien plus pratique, comparé au jeu d'instructions des anciens CPUs.

Si la condition est vraie (GT), alors la valeur de `D0` est copiée dans `D0` (i.e., il ne se passe rien). Si la condition n'est pas vraie, la valeur de `D1` est copiée dans `D0`.

GCC (Linaro) 4.9 sans optimisation

```
d_max:
; sauver les arguments en entrée dans la "Register Save Area"
; "zone de sauvegarde des registres"
        sub     sp, sp, #16
        str     d0, [sp,8]
        str     d1, [sp]
; recharger les valeurs
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        fmov    d0, x1
        fmov    d1, x0
; D0 - a, D1 - b
        fcmpe   d0, d1
        ble     .L76
; a>b; charger D0 (a) dans X0
        ldr     x0, [sp,8]
        b       .L74
.L76:
; a<=b; charger D1 (b) dans X0
        ldr     x0, [sp]
.L74:
; résultat dans X0
        fmov    d0, x0
; résultat dans D0
        add     sp, sp, 16
        ret
```

GCC sans optimisation est plus verbeux.

Tout d'abord, la fonction sauve la valeur de ses arguments en entrée dans la pile locale (*Register Save Area*, espace de sauvegarde des registres). Ensuite, le code recharge ces valeurs dans les registres X0/X1 et finalement les copie dans D0/D1 afin de les comparer en utilisant FCMPE. Beaucoup de code redondant, mais c'est ainsi que fonctionne les compilateurs sans optimisation. FCMPE compare les valeurs et met les flags du registre APSR. À ce moment, le compilateur ne pense pas encore à l'instruction plus commode FCSEL, donc il procède en utilisant de vieilles méthodes: en utilisant l'instruction BLE (*Branch if Less than or Equal* branchement si inférieur ou égal). Dans le premier cas ($a > b$), la valeur de a est chargée dans X0. Dans les autres cas ($a <= b$), la valeur de b est chargée dans X0. Enfin, la valeur dans X0 est copiée dans D0, car la valeur de retour doit être dans ce registre.

Exercice

À titre d'exercice, vous pouvez essayer d'optimiser ce morceau de code manuellement en supprimant les instructions redondantes et sans en introduire de nouvelles (incluant FCSEL).

GCC (Linaro) 4.9 avec optimisation—float

Ré-écrivons cet exemple en utilisant des *float* à la place de *double*.

```
float f_max (float a, float b)
{
        if (a>b)
                return a;

        return b;
};
```

```
f_max:
; S0 - a, S1 - b
        fcmpe   s0, s1
        fcsel   s0, s0, s1, gt
; maintenant le résultat est dans S0
        ret
```

C'est le même code, mais des S-registres sont utilisés à la place de D-registres. C'est parce que les nombres de type *float* sont passés dans des S-registres de 32-bit (qui sont en fait la partie basse des D-registres 64-bit).

MIPS

Le coprocesseur du processeur MIPS possède un bit de condition qui peut être mis par le FPU et lu par le CPU.

Les premiers MIPSs avaient seulement un bit de condition (appelé FCC0), les derniers en ont 8 (appelés FCC7-FCC0).

Ce bit (ou ces bits) sont situés dans un registre appelé FCCR.

Listing 1.226 : avec optimisation GCC 4.4.5 (IDA)

```
d_max:
; mettre le bit de condition du FPU si $f14<$f12 (b<a):
                c.lt.d  $f14, $f12
                or      $at, $zero ; NOP
; sauter en locret_14 si le bit de condition est mis
                bc1t    locret_14
; cette instruction est toujours exécutée (mettre la valeur de retour à "a"):
                mov.d   $f0, $f12  ; slot de délai de branchement
; cette instruction est exécutée seulement si la branche n'a pas été prise
; (i.e., si b>=a)
; mettre la valeur de retour à "b":
                mov.d   $f0, $f14

locret_14:
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

`C.LT.D` compare deux valeurs. LT est la condition «Less Than» (plus petit que). `D` implique des valeurs de type *double*. Suivant le résultat de la comparaison, le bit de condition FCC0 est mis à 1 ou à 0.

BC1T teste le bit FCC0 et saute si le bit est mis à 1. T signifie que le saut sera effectué si le bit est mis à 1 («True»). Il y a aussi une instruction BC1F qui saute si le bit n'est pas mis (donc est à 0) («False»).

Dépendant du saut, un des arguments de la fonction est placé dans $F0.

1.25.8 Quelques constantes

Il est facile de trouver la représentation de certaines constantes pour des nombres encodés au format IEEE 754 sur Wikipédia. Il est intéressant de savoir que 0,0 en IEEE 754 est représenté par 32 bits à zéro (pour la simple précision) ou 64 bits à zéro (pour la double). Donc pour mettre une variable flottante à 0,0 dans un registre ou en mémoire, on peut utiliser l'instruction MOV ou XOR reg, reg. Ceci est utilisable pour les structures où des variables de types variés sont présentes. Avec la fonction usuelle memset() il est possible de mettre toutes les variables entières à 0, toutes les variables booléennes à *false*, tous les pointeurs à NULL, et toutes les variables flottantes (de n'importe quelle précision) à 0,0.

1.25.9 Copie

On peut tout d'abord penser qu'il faut utiliser les instructions FLD/FST pour charger et stocker (et donc, copier) des valeurs IEEE 754. Néanmoins, la même chose peut-être effectuée plus facilement avec l'instruction usuelle MOV, qui, bien sûr, copie les valeurs au niveau binaire.

1.25.10 Pile, calculateurs et notation polonaise inverse

Maintenant nous comprenons pourquoi certains anciens calculateurs utilisent la notation Polonaise inverse.

Par exemple, pour additionner 12 et 34, on doit entrer 12, puis 34, et presser le signe «plus».

C'est parce que les anciens calculateurs programmable étaient simplement des implémentations de machine à pile, et c'était bien plus simple que de manipuler des expressions complexes avec parenthèses.

Un tel calculateur est encore présent dans de nombreuses distributions Unix: *dc*.

1.25.11 80 bits?

Représentation interne des nombres dans le FPU — 80-bit. Nombre étrange, car il n'est pas de la forme 2^n. Il y a une hypothèse que c'est probablement dû à des raisons historiques—le standard IBM de carte perforée peut encoder 12 lignes de 80 bits. La résolution en mode texte de $80 \cdot 25$ était aussi très populaire dans le passé.

Il y a une autre explication sur Wikipédia: https://en.wikipedia.org/wiki/Extended_precision.

Si vous en savez plus, s'il vous plaît, envoyez-moi un email: my emails.

1.25.12 x64

Sur la manière dont sont traités les nombres à virgules flottante en x86-64, lire ici: 1.38 on page 554.

1.25.13 Exercices

- http://challenges.re/60
- http://challenges.re/61

1.26 Tableaux

yy [120] Un tableau est simplement un ensemble de variables en mémoire qui sont situées les unes à côté des autres et qui ont le même type[121].

1.26.1 Exemple simple

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);

    return 0;
};
```

x86

MSVC

Compilons:

Listing 1.227 : MSVC 2008

```
_TEXT    SEGMENT
_i$ = -84                          ; size = 4
_a$ = -80                          ; size = 80
_main      PROC
    push   ebp
```

[120] AKA «homogener Container».
[121] AKA «container homogène»

```
    mov    ebp, esp
    sub    esp, 84        ; 00000054H
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN6@main
$LN5@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN6@main:
    cmp    DWORD PTR _i$[ebp], 20   ; 00000014H
    jge    SHORT $LN4@main
    mov    ecx, DWORD PTR _i$[ebp]
    shl    ecx, 1
    mov    edx, DWORD PTR _i$[ebp]
    mov    DWORD PTR _a$[ebp+edx*4], ecx
    jmp    SHORT $LN5@main
$LN4@main:
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 20    ; 00000014H
    jge    SHORT $LN1@main
    mov    ecx, DWORD PTR _i$[ebp]
    mov    edx, DWORD PTR _a$[ebp+ecx*4]
    push   edx
    mov    eax, DWORD PTR _i$[ebp]
    push   eax
    push   OFFSET $SG2463
    call   _printf
    add    esp, 12        ; 0000000cH
    jmp    SHORT $LN2@main
$LN1@main:
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

Rien de très particulier, juste deux boucles: la première est celle de remplissage et la seconde celle d'affichage. L'instruction `shl ecx, 1` est utilisée pour la multiplication par 2 de la valeur dans ECX, voir à ce sujet ci-après 1.24.2 on page 284.

80 octets sont alloués sur la pile pour le tableau, 20 éléments de 4 octets.

Essayons cet exemple dans OllyDbg.

Nous voyons comment le tableau est rempli:

chaque élément est un mot de 32-bit de type *int* et sa valeur est l'index multiplié par 2:

Fig. 1.87: OllyDbg : après remplissage du tableau

Puisque le tableau est situé sur la pile, nous pouvons voir ses 20 éléments ici.

GCC

Voici ce que GCC 4.4.1 génère:

Listing 1.228 : GCC 4.4.1

```
                public main
main            proc near                  ; DATA XREF: _start+17
```

```
var_70          = dword ptr -70h
var_6C          = dword ptr -6Ch
var_68          = dword ptr -68h
i_2             = dword ptr -54h
i               = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 70h
                mov     [esp+70h+i], 0           ; i=0
                jmp     short loc_804840A

loc_80483F7:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+70h+i]
                add     edx, edx                 ; edx=i*2
                mov     [esp+eax*4+70h+i_2], edx
                add     [esp+70h+i], 1           ; i++

loc_804840A:
                cmp     [esp+70h+i], 13h
                jle     short loc_80483F7
                mov     [esp+70h+i], 0
                jmp     short loc_8048441

loc_804841B:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+eax*4+70h+i_2]
                mov     eax, offset aADD ; "a[%d]=%d\n"
                mov     [esp+70h+var_68], edx
                mov     edx, [esp+70h+i]
                mov     [esp+70h+var_6C], edx
                mov     [esp+70h+var_70], eax
                call    _printf
                add     [esp+70h+i], 1

loc_8048441:
                cmp     [esp+70h+i], 13h
                jle     short loc_804841B
                mov     eax, 0
                leave
                retn
main            endp
```

À propos, la variable a est de type *int** (un pointeur sur un *int*)—vous pouvez passer un pointeur sur un tableau à une autre fonction, mais c'est plus juste de dire qu'un pointeur sur le premier élément du tableau est passé (les adresses du reste des éléments sont calculées de manière évidente).

Si vous indexez ce pointeur en *a[idx]*, il suffit d'ajouter *idx* au pointeur et l'élément placé ici (sur lequel pointe le pointeur calculé) est renvoyé.

Un exemple intéressant: une chaîne de caractères comme *string* est un tableau de caractères et a un type *const char[]*.

Un index peut aussi être appliqué à ce pointeur.

Et c'est pourquoi il est possible d'écrire des choses comme «string»[i]—c'est une expression C/C++ correcte!

ARM

sans optimisation Keil 6/2013 (Mode ARM)

```
            EXPORT _main
_main
            STMFD   SP!, {R4,LR}
            SUB     SP, SP, #0x50    ; allouer de la place pour 20 variables
    int

; première boucle

            MOV     R4, #0           ; i
            B       loc_4A0
loc_494
            MOV     R0, R4,LSL#1     ; R0=R4*2
            STR     R0, [SP,R4,LSL#2]; stocker R0 dans SP+R4<<2 (pareil que
    SP+R4*4)
            ADD     R4, R4, #1       ; i=i+1

loc_4A0
            CMP     R4, #20          ; i<20?
            BLT     loc_494          ; oui, effectuer encore le corps de la
    boucle

; seconde boucle

            MOV     R4, #0           ; i
            B       loc_4C4
loc_4B0
            LDR     R2, [SP,R4,LSL#2]; (second argument de printf)
    R2=*(SP+R4<<4)
                                     ; (pareil que *(SP+R4*4))
            MOV     R1, R4           ; (premier argument de printf) R1=i
            ADR     R0, aADD         ; "a[%d]=%d\n"
            BL      __2printf
            ADD     R4, R4, #1       ; i=i+1

loc_4C4
            CMP     R4, #20          ; i<20?
            BLT     loc_4B0          ; oui, effectuer encore le corps de la
    boucle
            MOV     R0, #0           ; valeur à renvoyer
            ADD     SP, SP, #0x50    ; libérer le chunk, alloué pour 20
    variables int
            LDMFD   SP!, {R4,PC}
```

Le type *int* nécessite 32 bits pour le stockage (ou 4 octets).

donc pour stocker 20 variables, *int* 80 (0x50) octets sont nécessaires.

C'est pourquoi l'instruction SUB SP, SP, #0x50 dans le prologue de la fonction alloue exactement cet espace sur la pile.

Dans la première et la seconde boucle, la variable de boucle *i* se trouve dans le registre R4.

Le nombre qui doit être écrit dans le tableau est calculé comme $i * 2$, ce qui est effectivement équivalent à décaler d'un bit vers la gauche, ce que fait l'instruction MOV R0, R4,LSL#1.

STR R0, [SP,R4,LSL#2] écrit le contenu de R0 dans le tableau.

Voici comment le pointeur sur un élément du tableau est calculé: SP pointe sur le début du tableau, R4 est i.

Donc décaler i de 2 bits vers la gauche est effectivement équivalent à la multiplication par 4 (puisque chaque élément du tableau a une taille de 4 octets) et ensuite on l'ajoute à l'adresse du début du tableau.

La seconde boucle a l'instruction inverse LDR R2, [SP,R4,LSL#2]. Elle charge la valeur du tableau dont nous avons besoin, et le pointeur est calculé de même.

avec optimisation Keil 6/2013 (Mode Thumb)

```
_main
          PUSH    {R4,R5,LR}
; allouer de l'espace pour 20 variables int + une variable supplémentaire
          SUB     SP, SP, #0x54

; première boucle

          MOVS    R0, #0      ; i
          MOV     R5, SP      ; pointeur sur le premier élément du tableau

loc_1CE
          LSLS    R1, R0, #1  ; R1=i<<1 (pareil que i*2)
          LSLS    R2, R0, #2  ; R2=i<<2 (pareil que i*4)
          ADDS    R0, R0, #1  ; i=i+1
          CMP     R0, #20     ; i<20?
          STR     R1, [R5,R2] ; stocker R1 dans *(R5+R2) (pareil que R5+i*4)
          BLT     loc_1CE     ; oui, i<20, effectuer encore le corps de la
    boucle

; seconde boucle

          MOVS    R4, #0      ; i=0
loc_1DC
          LSLS    R0, R4, #2  ; R0=i<<2 (pareil que i*4)
          LDR     R2, [R5,R0] ; charger depuis *(R5+R0) (pareil que R5+i*4)
          MOVS    R1, R4
          ADR     R0, aADD    ; "a[%d]=%d\n"
```

```
          BL      __2printf
          ADDS    R4, R4, #1  ; i=i+1
          CMP     R4, #20     ; i<20?
          BLT     loc_1DC     ; oui, i<20, effectuer encore le corps de la
    boucle
          MOVS    R0, #0      ; valeur à renvoyer
; libérer le chunk, alloué pour 20 variables int
; + une variable supplémentaire
          ADD     SP, SP, #0x54
          POP     {R4,R5,PC}
```

Le code Thumb est très similaire.

Le mode Thumb a des instructions spéciales pour le décalage (comme LSLS), qui calculent la valeur à écrire dans le tableau et l'adresse de chaque élément dans le tableau.

Le compilateur alloue légèrement plus d'espace sur la pile locale, cependant, les 4 derniers octets ne sont pas utilisés.

GCC 4.9.1 sans optimisation (ARM64)

Listing 1.229 : GCC 4.9.1 sans optimisation (ARM64)

```
.LC0:
        .string "a[%d]=%d\n"
main:
; sauver FR et LR sur la stack frame:
        stp     x29, x30, [sp, -112]!
; définir l'environnement de pile (FP=SP)
        add     x29, sp, 0
; initialiser la variable compteur à 0
; (WZR est le registre qui contient toujours zéro):
        str     wzr, [x29,108]
; sauter au code de test de condition de la boucle:
        b       .L2
.L3:
; charger la valeur de la variable "i":
        ldr     w0, [x29,108]
; la multiplier par 2:
        lsl     w2, w0, 1
; trouver la place du tableau dans la pile locale:
        add     x0, x29, 24
; charger l'entier 32-bit depuis la pile locale et l'étendre en un 64-bit
    signé:
        ldrsw   x1, [x29,108]
; calculer l'adresse de l'élément (X0+X1<<2=adresse du tableau+i*4)
; et y stocker W2 (i*2):
        str     w2, [x0,x1,lsl 2]
; incrémenter le compteur (i):
        ldr     w0, [x29,108]
        add     w0, w0, 1
        str     w0, [x29,108]
```

```
.L2:
; tester si la boucle est finie:
        ldr     w0, [x29,108]
        cmp     w0, 19
; sauter en L3 (début du corps de la boucle) si non:
        ble     .L3
; La seconde partie de la fonction commence ici.
; mettre la valeur initiale da la variable compteur à 0.
; à propos, le même espace est utilisé dans la pile locale,
; car la même variable locale (i) est utilisée comme compteur.
        str     wzr, [x29,108]
        b       .L4
.L5:
; calculer l'adresse dans le tableau:
        add     x0, x29, 24
; charger la valeur de "i":
        ldrsw   x1, [x29,108]
; charger la valeur du tableau à l'adresse (X0+X1<<2 = adresse du tableau +
    i*4)
        ldr     w2, [x0,x1,lsl 2]
; charger l'adresse de la chaîne "a[%d]=%d\n":
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; charger la variable "i" dans W1 et la passer à printf() comme second
    argument:
        ldr     w1, [x29,108]
; W2 contient toujours la valeur de l'élément du tableau qui vient d'être
    chargée.
; appeler printf():
        bl      printf
; incrémenter la variable "i":
        ldr     w0, [x29,108]
        add     w0, w0, 1
        str     w0, [x29,108]
.L4:
; est-ce fini?
        ldr     w0, [x29,108]
        cmp     w0, 19
; sauter au début du corps de la boucle si non:
        ble     .L5
; renvoyer 0
        mov     w0, 0
; restaurer FP et LR:
        ldp     x29, x30, [sp], 112
        ret
```

MIPS

La fonction utilise beaucoup de S- registres qui doivent être préservés, c'est pourquoi leurs valeurs sont sauvegardées dans la prologue de la fonction et restaurées dans l'épilogue.

Listing 1.230 : GCC 4.4.5 avec optimisation (IDA)

```
main:

var_70          = -0x70
var_68          = -0x68
var_14          = -0x14
var_10          = -0x10
var_C           = -0xC
var_8           = -8
var_4           = -4
; prologue de la fonction:
                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x80
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x80+var_4($sp)
                sw      $s3, 0x80+var_8($sp)
                sw      $s2, 0x80+var_C($sp)
                sw      $s1, 0x80+var_10($sp)
                sw      $s0, 0x80+var_14($sp)
                sw      $gp, 0x80+var_70($sp)
                addiu   $s1, $sp, 0x80+var_68
                move    $v1, $s1
                move    $v0, $zero
; cette valeur va être utilisée comme fin de boucle.
; elle a été pré-calculée par le compilateur GCC à l'étape de la compilation:
                li      $a0, 0x28  # '('

loc_34:                                  # CODE XREF: main+3C
; stocker la valeur en mémoire:
                sw      $v0, 0($v1)
; incrémenter la valeur à sauver de 2 à chaque itération:
                addiu   $v0, 2
; fin de boucle atteinte?
                bne     $v0, $a0, loc_34
; ajouter 4 à l'adresse dans tous les cas
                addiu   $v1, 4
; la boucle de remplissage du tableau est finie
; la seconde boucle commence
                la      $s3, $LC0        # "a[%d]=%d\n"
; la variable "i" va être dans $s0:
                move    $s0, $zero
                li      $s2, 0x14

loc_54:                                  # CODE XREF: main+70
; appeler printf():
                lw      $t9, (printf & 0xFFFF)($gp)
                lw      $a2, 0($s1)
                move    $a1, $s0
                move    $a0, $s3
                jalr    $t9
; incrémenter "i":
                addiu   $s0, 1
                lw      $gp, 0x80+var_70($sp)
; sauter au corps de la boucle si la fin n'est pas atteinte:
```

```
                bne     $s0, $s2, loc_54
; déplacer le pointeur mémoire au prochain mot de 32-bit:
                addiu   $s1, 4
; épilogue de la fonction
                lw      $ra, 0x80+var_4($sp)
                move    $v0, $zero
                lw      $s3, 0x80+var_8($sp)
                lw      $s2, 0x80+var_C($sp)
                lw      $s1, 0x80+var_10($sp)
                lw      $s0, 0x80+var_14($sp)
                jr      $ra
                addiu   $sp, 0x80

$LC0:           .ascii "a[%d]=%d\n"<0>   # DATA XREF: main+44
```

Quelque chose d'intéressant: il y a deux boucles et la première n'a pas besoin de i, elle a seulement besoin de $i * 2$ (augmenté de 2 à chaque itération) et aussi de l'adresse en mémoire (augmentée de 4 à chaque itération).

Donc ici nous voyons deux variables, une (dans $V0) augmentée de 2 à chaque fois, et une autre (dans $V1) — de 4.

La seconde boucle est celle où `printf()` est appelée et affiche la valeur de i à l'utilisateur, donc il y a une variable qui est incrémentée de 1 à chaque fois (dans $S0) et aussi l'adresse en mémoire (dans $S1) incrémentée de 4 à chaque fois.

Cela nous rappelle l'optimisation de boucle que nous avons examiné avant: 3.10 on page 627.

Leur but est de se passer des multiplications.

1.26.2 Débordement de tampon

Lire en dehors des bornes du tableau

Donc, indexer un tableau est juste *array[index]*. Si vous étudiez le code généré avec soin, vous remarquerez sans doute l'absence de test sur les bornes de l'index, qui devrait vérifier *si il est inférieur à 20*. Que ce passe-t-il si l'index est supérieur à 20? C'est une des caractéristiques de C/C++ qui est souvent critiquée.

Voici un code qui compile et fonctionne:

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<20; i++)
                a[i]=i*2;

        printf ("a[20]=%d\n", a[20]);
```

```
        return 0;
};
```

Résultat de la compilation (MSVC 2008) :

Listing 1.231 : MSVC 2008 sans optimisation

```
$SG2474 DB    'a[20]=%d', 0aH, 00H

_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 20
    jge    SHORT $LN1@main
    mov    ecx, DWORD PTR _i$[ebp]
    shl    ecx, 1
    mov    edx, DWORD PTR _i$[ebp]
    mov    DWORD PTR _a$[ebp+edx*4], ecx
    jmp    SHORT $LN2@main
$LN1@main:
    mov    eax, DWORD PTR _a$[ebp+80]
    push   eax
    push   OFFSET $SG2474 ; 'a[20]=%d'
    call   DWORD PTR __imp__printf
    add    esp, 8
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

Le code produit ce résultat:

Listing 1.232 : OllyDbg : sortie sur la console

```
a[20]=1638280
```

C’est juste *quelque chose* qui se trouvait sur la pile à côté du tableau, 80 octets après le début de son premier élément.

Essayons de trouver d'où vient cette valeur, en utilisant OllyDbg.

Chargeons et trouvons la valeur située juste après le dernier élément du tableau:

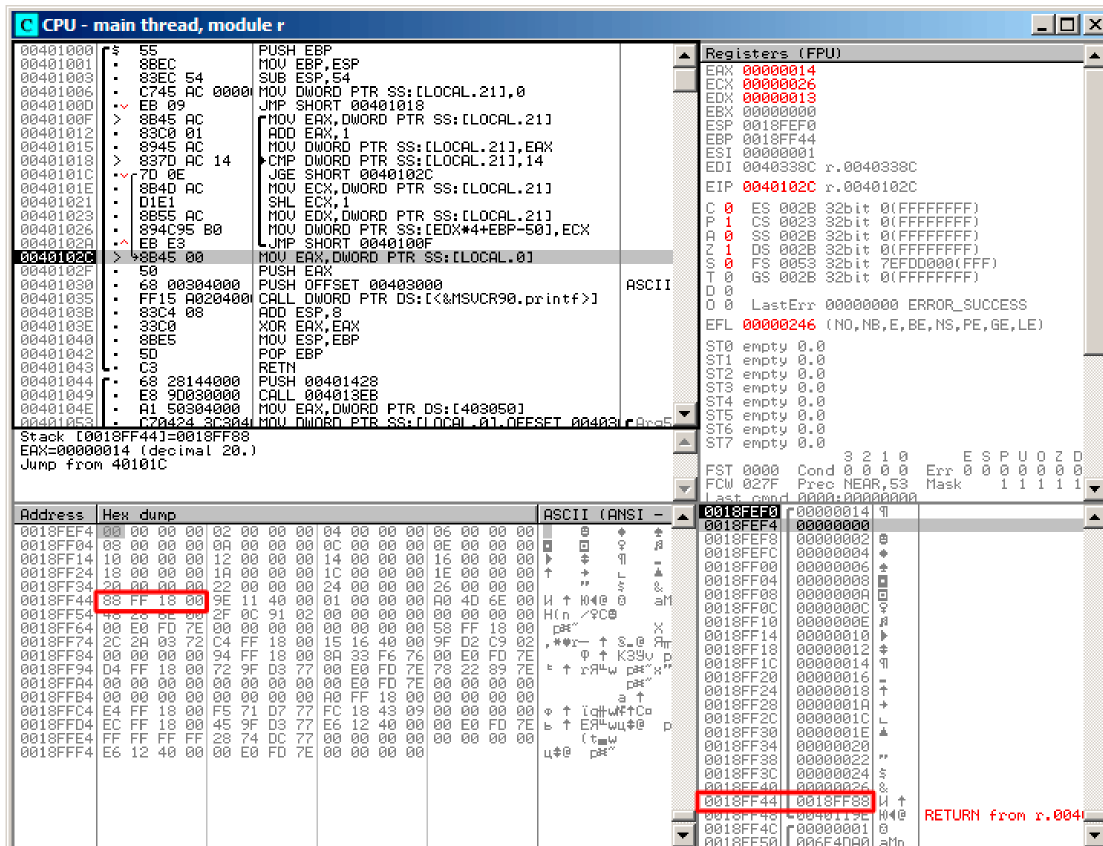


Fig. 1.88: OllyDbg : lecture du 20ème élément et exécution de `printf()`

Qu'est-ce que c'est? D'après le schéma de la pile, c'est la valeur sauvegardée du registre EBP.

Exécutons encore et voyons comment il est restauré:

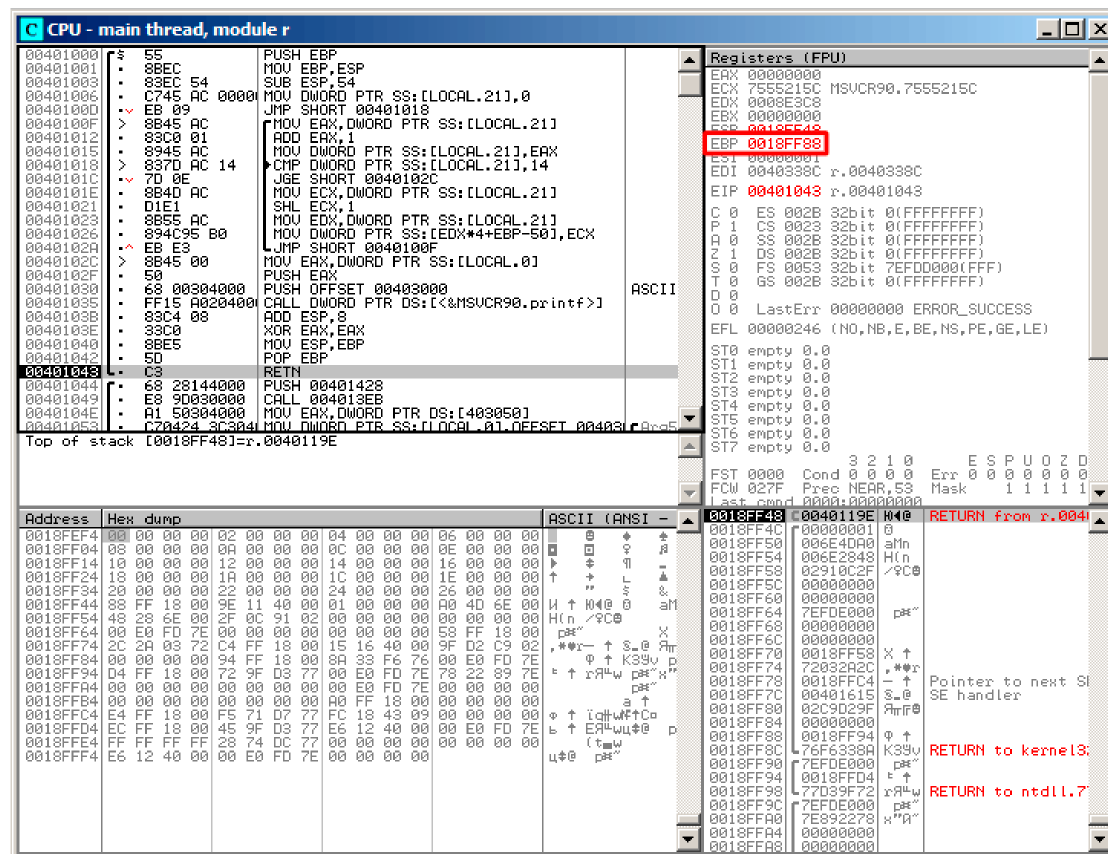


Fig. 1.89: OllyDbg : restaurer la valeur de EBP

En effet, comment est-ce ça pourrait être différent? Le compilateur pourrait générer du code supplémentaire pour vérifier que la valeur de l'index est toujours entre les bornes du tableau (comme dans les langages de programmation de plus haut-niveau[122]) mais cela rendrait le code plus lent.

Écrire hors des bornes du tableau

Ok, nous avons lu quelques valeurs de la pile *illégalement*, mais que se passe-t-il si nous essayons d'écrire quelque chose?

Voici ce que nous avons:

```
#include <stdio.h>

int main()
```

[122] Java, Python, etc.

```
{
        int a[20];
        int i;

        for (i=0; i<30; i++)
                a[i]=i;

        return 0;
};
```

MSVC

Et ce que nous obtenons:

Listing 1.233 : MSVC 2008 sans optimisation

```
_TEXT    SEGMENT
_i$ = -84 ; taille = 4
_a$ = -80 ; taille = 80
_main   PROC
 push   ebp
 mov    ebp, esp
 sub    esp, 84
 mov    DWORD PTR _i$[ebp], 0
 jmp    SHORT $LN3@main
$LN2@main:
 mov    eax, DWORD PTR _i$[ebp]
 add    eax, 1
 mov    DWORD PTR _i$[ebp], eax
$LN3@main:
 cmp    DWORD PTR _i$[ebp], 30 ; 0000001eH
 jge    SHORT $LN1@main
 mov    ecx, DWORD PTR _i$[ebp]
 mov    edx, DWORD PTR _i$[ebp]      ; cette instruction est évidemment
    redondante
 mov    DWORD PTR _a$[ebp+ecx*4], edx ; ECX pourrait être utilisé en second
    opérande ici
 jmp    SHORT $LN2@main
$LN1@main:
 xor    eax, eax
 mov    esp, ebp
 pop    ebp
 ret    0
_main   ENDP
```

Le programme compilé plante après le lancement. Pas de miracle. Voyons exactement où il plante.

Chargeons le dans OllyDbg, et traçons le jusqu'à ce que les 30 éléments du tableau soient écrits:

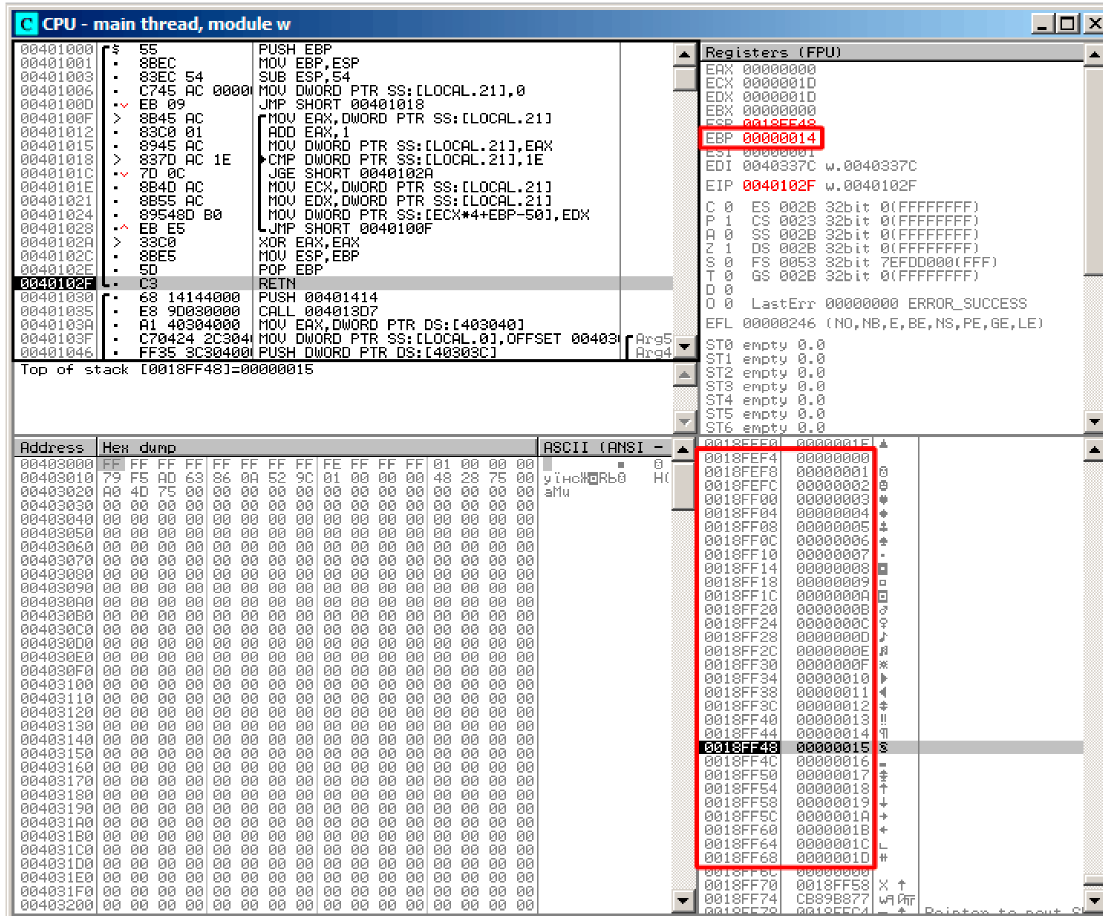


Fig. 1.90: OllyDbg : après avoir restauré la valeur de EBP

Exécutons pas à pas jusqu'à la fin de la fonction:

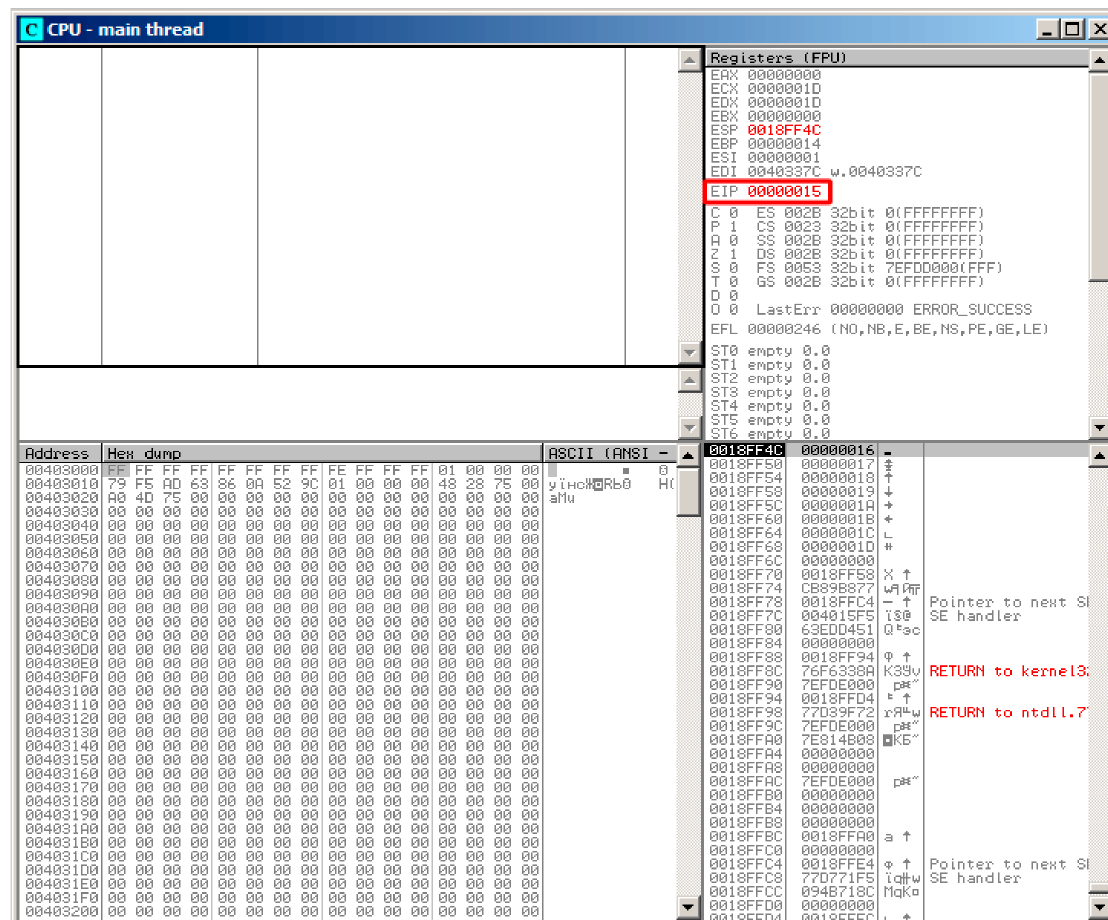


Fig. 1.91: OllyDbg : EIP a été restauré, mais OllyDbg ne peut pas désassembler en 0x15

Maintenant, gardez vos yeux sur les registres.

EIP contient maintenant 0x15. Ce n'est pas une adresse légale pour du code—au moins pour du code win32! Nous sommes arrivés ici contre notre volonté. Il est aussi intéressant de voir que le registre EBP contient 0x14, ECX et EDX contiennent 0x1D.

Étudions un peu plus la structure de la pile.

Après que le contrôle du flux a été passé à `main()`, la valeur du registre EBP a été sauvée sur la pile. Puis, 84 octets ont été alloués pour le tableau et la variable i. C'est `(20+1)*sizeof(int)`. ESP pointe maintenant sur la variable `_i` dans la pile locale et après l'exécution du PUSH `quelquechose` suivant, *quelquechose* apparaît à côté de `_i`.

C'est la structure de la pile pendant que le contrôle est dans `main()` :

ESP	4 octets alloués pour la variable i
ESP+4	80 octets alloués pour le tableau a[20]
ESP+84	valeur sauvegardée de EBP
ESP+88	adresse de retour

L'expression a[19]=quelquechose écrit le dernier *int* dans des bornes du tableau (dans les limites jusqu'ici!)

L'expression a[20]=quelquechose écrit *quelquechose* à l'endroit où la valeur sauvegardée de EBP se trouve.

S'il vous plaît, regardez l'état du registre lors du plantage. Dans notre cas, 20 a été écrit dans le 20ème élément. À la fin de la fonction, l'épilogue restaure la valeur d'origine de EBP. (20 en décimal est 0x14 en hexadécimal). Ensuite RET est exécuté, qui est équivalent à l'instruction POP EIP.

L'instruction RET prend la valeur de retour sur la pile (c'est l'adresse dans CRT), qui a appelé main()), et 21 est stocké ici (0x15 en hexadécimal). Le CPU trape à l'adresse 0x15, mais il n'y a pas de code exécutable ici, donc une exception est levée.

Bienvenu! Ça s'appelle un *buffer overflow (débordement de tampon)*[123].

Remplacez la tableau de *int* avec une chaîne (*char* array), créez délibérément une longue chaîne et passez-là au programme, à la fonction, qui ne teste pas la longueur de la chaîne et la copie dans un petit buffer et vous serez capable de faire pointer le programme à une adresse où il devra sauter. C'est pas aussi simple dans la réalité, mais c'est comme cela que ça a apparu. L'article classique à propos de ça: [Aleph One, *Smashing The Stack For Fun And Profit*, (1996)][124].

GCC

Essayons le même code avec GCC 4.4.1. Nous obtenons:

```
                public main
main            proc near

a               = dword ptr -54h
i               = dword ptr -4

                push    ebp
                mov     ebp, esp
                sub     esp, 60h ; 96
                mov     [ebp+i], 0
                jmp     short loc_80483D1
loc_80483C3:
                mov     eax, [ebp+i]
                mov     edx, [ebp+i]
                mov     [ebp+eax*4+a], edx
                add     [ebp+i], 1
loc_80483D1:
```

[123] Wikipédia
[124] Aussi disponible en http://yurichev.com/mirrors/phrack/p49-0x0e.txt

```
                cmp     [ebp+i], 1Dh
                jle     short loc_80483C3
                mov     eax, 0
                leave
                retn
main            endp
```

Lancer ce programme sous Linux donnera: `Segmentation fault`.

Si nous le lançons dans le débogueur GDB, nous obtenons ceci:

```
(gdb) r
Starting program: /home/dennis/RE/1

Program received signal SIGSEGV, Segmentation fault.
0x00000016 in ?? ()
(gdb) info registers
eax            0x0      0
ecx            0xd2f96388       -755407992
edx            0x1d     29
ebx            0x26eff4 2551796
esp            0xbffff4b0       0xbffff4b0
ebp            0x15     0x15
esi            0x0      0
edi            0x0      0
eip            0x16     0x16
eflags         0x10202  [ IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

Les valeurs des registres sont légèrement différentes de l'exemple win32, puisque la structure de la pile est également légèrement différente.

1.26.3 Méthodes de protection contre les débordements de tampon

Il existe quelques méthodes pour protéger contre ce fléau, indépendamment de la négligence des programmeurs C/C++. MSVC possède des options comme[125] :

```
 /RTCs Stack Frame runtime checking
 /GZ Enable stack checks (/RTCs)
```

Une des méthodes est d'écrire une valeur aléatoire entre les variables locales sur la pile dans le prologue de la fonction et de la vérifier dans l'épilogue, avant de sortir

[125] méthode de protection contre les débordements de tampons côté compilateur:wikipedia.org/wiki/Buffer_overflow_protection

de la fonction. Si la valeur n'est pas la même, ne pas exécuter la dernière instruction RET, mais stopper (ou bloquer). Le processus va s'arrêter, mais c'est mieux qu'une attaque distante sur votre ordinateur.

Cette valeur aléatoire est parfois appelé un «canari», c'est lié au canari[126] que les mineurs utilisaient dans le passé afin de détecter rapidement les gaz toxiques.

Les canaris sont très sensibles aux gaz, ils deviennent très agités en cas de danger, et même meurent.

Si nous compilons notre exemple de tableau très simple (1.26.1 on page 341) dans MSVC avec les options RTC1 et RTCs, nous voyons un appel à `@_RTC_CheckStackVars@8` une fonction à la fin de la fonction qui vérifie si le «canari» est correct.

Voyons comment GCC gère ceci. Prenons un exemple `alloca()` (1.9.2 on page 49) :

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

Par défaut, sans option supplémentaire, GCC 4.7.3 insère un test de «canari» dans le code:

Listing 1.234 : GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 676
        lea     ebx, [esp+39]
        and     ebx, -16
        mov     DWORD PTR [esp+20], 3
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0  ; "hi! %d, %d, %d\n"
```

[126] wikipedia.org/wiki/Domestic_canary#Miner.27s_canary

```
        mov     DWORD PTR [esp+4], 600
        mov     DWORD PTR [esp], ebx
        mov     eax, DWORD PTR gs:20     ; canari
        mov     DWORD PTR [ebp-12], eax
        xor     eax, eax
        call    _snprintf
        mov     DWORD PTR [esp], ebx
        call    puts
        mov     eax, DWORD PTR [ebp-12]
        xor     eax, DWORD PTR gs:20     ; teste le canari
        jne     .L5
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
.L5:
        call    __stack_chk_fail
```

La valeur aléatoire se trouve en gs:20. Elle est écrite sur la pile et à la fin de la fonction, la valeur sur la pile est comparée avec le «canari» correct dans gs:20. Si les valeurs ne sont pas égales, la fonction __stack_chk_fail est appelée et nous voyons dans la console quelque chose comme ça (Ubuntu 13.04 x86) :

```
*** buffer overflow detected ***: ./2_1 terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x63)[0xb7699bc3]
/lib/i386-linux-gnu/libc.so.6(+0x10593a)[0xb769893a]
/lib/i386-linux-gnu/libc.so.6(+0x105008)[0xb7698008]
/lib/i386-linux-gnu/libc.so.6(_IO_default_xsputn+0x8c)[0xb7606e5c]
/lib/i386-linux-gnu/libc.so.6(_IO_vfprintf+0x165)[0xb75d7a45]
/lib/i386-linux-gnu/libc.so.6(__vsprintf_chk+0xc9)[0xb76980d9]
/lib/i386-linux-gnu/libc.so.6(__sprintf_chk+0x2f)[0xb7697fef]
./2_1[0x8048404]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf5)[0xb75ac935]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 2097586    /home/dennis/2_1
08049000-0804a000 r--p 00000000 08:01 2097586    /home/dennis/2_1
0804a000-0804b000 rw-p 00001000 08:01 2097586    /home/dennis/2_1
094d1000-094f2000 rw-p 00000000 00:00 0          [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602    /lib/i386-linux-gnu/↙
    ↳ libgcc_s.so.1
b757b000-b757c000 r--p 0001a000 08:01 1048602    /lib/i386-linux-gnu/↙
    ↳ libgcc_s.so.1
b757c000-b757d000 rw-p 0001b000 08:01 1048602    /lib/i386-linux-gnu/↙
    ↳ libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781    /lib/i386-linux-gnu/libc↙
    ↳ -2.17.so
b7740000-b7742000 r--p 001ad000 08:01 1050781    /lib/i386-linux-gnu/libc↙
    ↳ -2.17.so
b7742000-b7743000 rw-p 001af000 08:01 1050781    /lib/i386-linux-gnu/libc↙
    ↳ -2.17.so
b7743000-b7746000 rw-p 00000000 00:00 0
b775a000-b775d000 rw-p 00000000 00:00 0
```

```
b775d000-b775e000 r-xp 00000000 00:00 0          [vdso]
b775e000-b777e000 r-xp 00000000 08:01 1050794    /lib/i386-linux-gnu/ld↙
    ↳ -2.17.so
b777e000-b777f000 r--p 0001f000 08:01 1050794    /lib/i386-linux-gnu/ld↙
    ↳ -2.17.so
b777f000-b7780000 rw-p 00020000 08:01 1050794    /lib/i386-linux-gnu/ld↙
    ↳ -2.17.so
bff35000-bff56000 rw-p 00000000 00:00 0          [stack]
Aborted (core dumped)
```

gs est ainsi appelé registre de segment. Ces registres étaient beaucoup utilisés du temps de MS-DOS et des extensions de DOS. Aujourd'hui, sa fonction est différente.

Dit brièvement, le registre gs dans Linux pointe toujours sur le TLS[127] (6.2 on page 965)—des informations spécifiques au thread sont stockées là. À propos, en win32 le registre fs joue le même rôle, pointant sur TIB[128] [129].

Il y a plus d'information dans le code source du noyau Linux (au moins dans la version 3.11), dans
arch/x86/include/asm/stackprotector.h cette variable est décrite dans les commentaires.

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

Reprenons notre exemple de simple tableau (1.26.1 on page 341),

à nouveau, nous pouvons voir comment LLVM teste si le «canari» est correct:

```
_main

var_64          = -0x64
var_60          = -0x60
var_5C          = -0x5C
var_58          = -0x58
var_54          = -0x54
var_50          = -0x50
var_4C          = -0x4C
var_48          = -0x48
var_44          = -0x44
var_40          = -0x40
var_3C          = -0x3C
var_38          = -0x38
var_34          = -0x34
var_30          = -0x30
var_2C          = -0x2C
var_28          = -0x28
var_24          = -0x24
var_20          = -0x20
var_1C          = -0x1C
```

[127]Thread Local Storage
[128]Thread Information Block
[129]wikipedia.org/wiki/Win32_Thread_Information_Block

```
var_18          = -0x18
canary          = -0x14
var_10          = -0x10

    PUSH     {R4-R7,LR}
    ADD      R7, SP, #0xC
    STR.W    R8, [SP,#0xC+var_10]!
    SUB      SP, SP, #0x54
    MOVW     R0, #aObjc_methtype ; "objc_methtype"
    MOVS     R2, #0
    MOVT.W   R0, #0
    MOVS     R5, #0
    ADD      R0, PC
    LDR.W    R8, [R0]
    LDR.W    R0, [R8]
    STR      R0, [SP,#0x64+canari]
    MOVS     R0, #2
    STR      R2, [SP,#0x64+var_64]
    STR      R0, [SP,#0x64+var_60]
    MOVS     R0, #4
    STR      R0, [SP,#0x64+var_5C]
    MOVS     R0, #6
    STR      R0, [SP,#0x64+var_58]
    MOVS     R0, #8
    STR      R0, [SP,#0x64+var_54]
    MOVS     R0, #0xA
    STR      R0, [SP,#0x64+var_50]
    MOVS     R0, #0xC
    STR      R0, [SP,#0x64+var_4C]
    MOVS     R0, #0xE
    STR      R0, [SP,#0x64+var_48]
    MOVS     R0, #0x10
    STR      R0, [SP,#0x64+var_44]
    MOVS     R0, #0x12
    STR      R0, [SP,#0x64+var_40]
    MOVS     R0, #0x14
    STR      R0, [SP,#0x64+var_3C]
    MOVS     R0, #0x16
    STR      R0, [SP,#0x64+var_38]
    MOVS     R0, #0x18
    STR      R0, [SP,#0x64+var_34]
    MOVS     R0, #0x1A
    STR      R0, [SP,#0x64+var_30]
    MOVS     R0, #0x1C
    STR      R0, [SP,#0x64+var_2C]
    MOVS     R0, #0x1E
    STR      R0, [SP,#0x64+var_28]
    MOVS     R0, #0x20
    STR      R0, [SP,#0x64+var_24]
    MOVS     R0, #0x22
    STR      R0, [SP,#0x64+var_20]
    MOVS     R0, #0x24
    STR      R0, [SP,#0x64+var_1C]
```

```
    MOVS     R0, #0x26
    STR      R0, [SP,#0x64+var_18]
    MOV      R4, 0xFDA ; "a[%d]=%d\n"
    MOV      R0, SP
    ADDS     R6, R0, #4
    ADD      R4, PC
    B        loc_2F1C

; début de la seconde boucle

loc_2F14
    ADDS     R0, R5, #1
    LDR.W    R2, [R6,R5,LSL#2]
    MOV      R5, R0

loc_2F1C
    MOV      R0, R4
    MOV      R1, R5
    BLX      _printf
    CMP      R5, #0x13
    BNE      loc_2F14
    LDR.W    R0, [R8]
    LDR      R1, [SP,#0x64+canari]
    CMP      R0, R1
    ITTTT EQ            ; est-ce que le canari est toujours correct?
    MOVEQ    R0, #0
    ADDEQ    SP, SP, #0x54
    LDREQ.W  R8, [SP+0x64+var_64],#4
    POPEQ    {R4-R7,PC}
    BLX      ___stack_chk_fail
```

Tout d'abord, on voit que LLVM a «déroulé» la boucle et que toutes les valeurs sont écrites une par une, pré-calculée, car LLVM a conclu que c'est plus rapide. À propos, des instructions en mode ARM peuvent aider à rendre cela encore plus rapide, et les trouver peut être un exercice pour vous.

À la fin de la fonction, nous voyons la comparaison des «canaris»—celui sur la pile locale et le correct.

S'ils sont égaux, un bloc de 4 instructions est exécuté par `ITTTT EQ`, qui contient l'écriture de 0 dans `R0`, l'épilogue de la fonction et la sortie. Si les «canaris» ne sont pas égaux, le bloc est passé, et la fonction saute en `___stack_chk_fail`, qui, peut-être, stoppe l'exécution.

1.26.4 Encore un mot sur les tableaux

Maintenant nous comprenons pourquoi il est impossible d'écrire quelque chose comme ceci en code C/C++ :

```
void f(int size)
{
    int a[size];
...
```

```
};
```

C’est simplement parce que le compilateur doit connaître la taille exacte du tableau pour lui allouer de l’espace sur la pile locale lors de l’étape de compilation.

Si vous avez besoin d’un tableau de taille arbitraire, il faut l’allouer en utilisant `malloc()`, puis en accédant aux blocs de mémoire allouée comme un tableau de variables du type dont vous avez besoin.

Ou utiliser la caractéristique du standart C99 [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.5/2], et qui fonctionne comme *alloca()* (1.9.2 on page 49) en interne.

Il est aussi possible d’utiliser des bibliothèques de ramasse-miettes pour C.

Et il y a aussi des bibliothèques supportant les pointeurs intelligents pour C++.

1.26.5 Tableau de pointeurs sur des chaînes

Voici un exemple de tableau de pointeurs.[130]

Listing 1.235 : Prendre le nom du mois

```
#include <stdio.h>

const char* month1[]=
{
        "janvier", "fevrier", "mars", "avril",
        "mai", "juin", "juillet", "aout",
        "septembre", "octobre", "novembre", "decembre"
};

// dans l'intervalle 0..11
const char* get_month1 (int month)
{
        return month1[month];
};
```

x64

Listing 1.236 : MSVC 2013 avec optimisation x64

```
_DATA   SEGMENT
month1  DQ      FLAT:$SG3122
        DQ      FLAT:$SG3123
        DQ      FLAT:$SG3124
        DQ      FLAT:$SG3125
        DQ      FLAT:$SG3126
        DQ      FLAT:$SG3127
        DQ      FLAT:$SG3128
        DQ      FLAT:$SG3129
```

[130]NDT: attention à l’encodage des fichiers, en ASCII ou en ISO-8859, un caractère occupe un octet, alors qu’en UTF-8, notamment, il peut en occuper plusieurs. Par exemple, ’û’ est codé $fb (1 octet) en ISO-8859 et $c3$bb (2 octets) en UTF-8. J’ai donc volontairement mis des caractères non accentués dans le code.

```
        DQ      FLAT:$SG3130
        DQ      FLAT:$SG3131
        DQ      FLAT:$SG3132
        DQ      FLAT:$SG3133
$SG3122 DB      'January', 00H
$SG3123 DB      'February', 00H
$SG3124 DB      'March', 00H
$SG3125 DB      'April', 00H
$SG3126 DB      'May', 00H
$SG3127 DB      'June', 00H
$SG3128 DB      'July', 00H
$SG3129 DB      'August', 00H
$SG3130 DB      'September', 00H
$SG3156 DB      '%s', 0aH, 00H
$SG3131 DB      'October', 00H
$SG3132 DB      'November', 00H
$SG3133 DB      'December', 00H
_DATA   ENDS

month$ = 8
get_month1 PROC
        movsxd  rax, ecx
        lea     rcx, OFFSET FLAT:month1
        mov     rax, QWORD PTR [rcx+rax*8]
        ret     0
get_month1 ENDP
```

Le code est très simple:

- La première instruction MOVSXD copie une valeur 32-bit depuis ECX (où l'argument $month$ est passé) dans RAX avec extension du signe (car l'argument $month$ est de type *int*).

  La raison de l'extension du signe est que cette valeur 32-bit va être utilisée dans des calculs avec d'autres valeurs 64-bit.

  C'est pourquoi il doit être étendu à 64-bit[131].

- Ensuite l'adresse du pointeur de la table est chargée dans RCX.

- Enfin, la valeur d'entrée ($month$) est multipliée par 8 et ajoutée à l'adresse. Effectivement: nous sommes dans un environnement 64-bit et toutes les adresses (ou pointeurs) nécessitent exactement 64 bits (ou 8 octets) pour être stockées. C'est pourquoi chaque élément de la table a une taille de 8 octets. Et c'est pourquoi pour prendre un élément spécifique, $month * 8$ octets doivent être passés depuis le début. C'est ce que fait MOV. De plus, cette instruction charge également l'élément à cette adresse. Pour 1, l'élément sera un pointeur sur la chaîne qui contient «février», etc.

[131] C'est parfois bizarre, mais des indices négatifs de tableau peuvent être passés par $month$ (les indices négatifs de tableaux sont expliqués plus loin: 3.22 on page 769). Et si cela arrive, la valeur entrée négative de type *int* est étendue correctement et l'élément correspondant avant le tableau est sélectionné. Ça ne fonctionnera pas correctement sans l'extension du signe.

GCC 4.9 avec optimisation peut faire encore mieux[132] :

Listing 1.237 : GCC 4.9 avec optimisation x64

```
        movsx   rdi, edi
        mov     rax, QWORD PTR month1[0+rdi*8]
        ret
```

MSVC 32-bit

Compilons-le aussi avec le compilateur MSVC 32-bit:

Listing 1.238 : MSVC 2013 avec optimisation x86

```
_month$ = 8
_get_month1 PROC
        mov     eax, DWORD PTR _month$[esp-4]
        mov     eax, DWORD PTR _month1[eax*4]
        ret     0
_get_month1 ENDP
```

La valeur en entrée n'a pas besoin d'être étendue sur 64-bit, donc elle est utilisée telle quelle.

Et elle est multipliée par 4, car les éléments de la table sont larges de 32-bit (ou 4 octets).

ARM 32-bit

ARM en mode ARM

Listing 1.239 : avec optimisation Keil 6/2013 (Mode ARM)

```
get_month1 PROC
        LDR      r1,|L0.100|
        LDR      r0,[r1,r0,LSL #2]
        BX       lr
        ENDP

|L0.100|
        DCD      ||.data||

        DCB      "January",0
        DCB      "February",0
        DCB      "March",0
        DCB      "April",0
        DCB      "May",0
        DCB      "June",0
        DCB      "July",0
```

[132] «0+ » a été laissé dans le listing car la sortie de l'assembleur GCC n'est pas assez soignée pour l'éliminer. C'est un *déplacement*, et il vaut zéro ici.

```
        DCB      "August",0
        DCB      "September",0
        DCB      "October",0
        DCB      "November",0
        DCB      "December",0

        AREA ||.data||, DATA, ALIGN=2
month1
        DCD      ||.conststring||
        DCD      ||.conststring||+0x8
        DCD      ||.conststring||+0x11
        DCD      ||.conststring||+0x17
        DCD      ||.conststring||+0x1d
        DCD      ||.conststring||+0x21
        DCD      ||.conststring||+0x26
        DCD      ||.conststring||+0x2b
        DCD      ||.conststring||+0x32
        DCD      ||.conststring||+0x3c
        DCD      ||.conststring||+0x44
        DCD      ||.conststring||+0x4d
```

L'adresse de la table est chargée en R1.

Tout le reste est effectué en utilisant juste une instruction LDR.

Puis la valeur en entrée est décalée de 2 vers la gauche (ce qui est la même chose que multiplier par 4), puis ajoutée à R1 (où se trouve l'adresse de la table) et enfin un élément de la table est chargé depuis cette adresse.

L'élément 32-bit de la table est chargé dans R1 depuis la table.

ARM en mode Thumb

Le code est essentiellement le même, mais moins dense, car le suffixe LSL ne peut pas être spécifié dans l'instruction LDR ici:

```
get_month1 PROC
        LSLS     r0,r0,#2
        LDR      r1,|L0.64|
        LDR      r0,[r1,r0]
        BX       lr
        ENDP
```

ARM64

Listing 1.240 : GCC 4.9 avec optimisation ARM64

```
get_month1:
        adrp    x1, .LANCHOR0
        add     x1, x1, :lo12:.LANCHOR0
        ldr     x0, [x1,w0,sxtw 3]
        ret
```

```
.LANCHOR0 = . + 0
        .type   month1, %object
        .size   month1, 96
month1:
        .xword  .LC2
        .xword  .LC3
        .xword  .LC4
        .xword  .LC5
        .xword  .LC6
        .xword  .LC7
        .xword  .LC8
        .xword  .LC9
        .xword  .LC10
        .xword  .LC11
        .xword  .LC12
        .xword  .LC13
.LC2:
        .string "January"
.LC3:
        .string "February"
.LC4:
        .string "March"
.LC5:
        .string "April"
.LC6:
        .string "May"
.LC7:
        .string "June"
.LC8:
        .string "July"
.LC9:
        .string "August"
.LC10:
        .string "September"
.LC11:
        .string "October"
.LC12:
        .string "November"
.LC13:
        .string "December"
```

L'adresse de la table est chargée dans X1 en utilisant la paire ADRP/ADD.

Puis l'élément correspondant est choisi dans la table en utilisant seulement un LDR, qui prend W0 (le registre où l'argument d'entrée $month$ se trouve), le décale de 3 bits vers la gauche (ce qui est la même chose que de le multiplier par 8), étend son signe (c'est ce que le suffixe «sxtw» implique) et l'ajoute à X0. Enfin la valeur 64-bit est chargée depuis la table dans X0.

MIPS

Listing 1.241 : GCC 4.4.5 avec optimisation (IDA)

```
get_month1:
; charger l'adresse de la table dans $v0:
                la      $v0, month1
; prendre la valeur en entrée et la multiplier par 4:
                sll     $a0, 2
; ajouter l'adresse de la table et la valeur multipliée:
                addu    $a0, $v0
; charger l'élément de la table à cette adresse dans $v0:
                lw      $v0, 0($a0)
; sortir
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP

                .data # .data.rel.local
                .globl month1
month1:         .word aJanuary           # "janvier"
                .word aFebruary          # "fevrier"
                .word aMarch             # "mars"
                .word aApril             # "avril"
                .word aMay               # "mai"
                .word aJune              # "juin"
                .word aJuly              # "juillet"
                .word aAugust            # "aout"
                .word aSeptember         # "septembre"
                .word aOctober           # "octobre"
                .word aNovember          # "novembre"
                .word aDecember          # "decembre"

                .data # .rodata.str1.4
aJanuary:       .ascii "janvier"<0>
aFebruary:      .ascii "fevrier"<0>
aMarch:         .ascii "mars"<0>
aApril:         .ascii "avril"<0>
aMay:           .ascii "mai"<0>
aJune:          .ascii "juin"<0>
aJuly:          .ascii "juillet"<0>
aAugust:        .ascii "aout"<0>
aSeptember:     .ascii "septembre"<0>
aOctober:       .ascii "octobre"<0>
aNovember:      .ascii "novembre"<0>
aDecember:      .ascii "decembre"<0>
```

Débordement de tableau

Notre fonction accepte des valeurs dans l'intervalle 0..11, mais que se passe-t-il si 12 est passé? Il n'y a pas d'élément dans la table à cet endroit.

Donc la fonction va charger la valeur qui se trouve là, et la renvoyer.

Peu après, une autre fonction pourrait essayer de lire une chaîne de texte depuis cette adresse et pourrait planter.

Compilons l'exemple dans MSVC pour win64 et ouvrons le dans IDA pour voir ce que l'éditeur de lien à stocker après la table:

Listing 1.242 : Fichier exécutable dans IDA

```
off_140011000   dq offset aJanuary_1    ; DATA XREF: .text:0000000140001003
                                        ; "January"
                dq offset aFebruary_1   ; "February"
                dq offset aMarch_1      ; "March"
                dq offset aApril_1      ; "April"
                dq offset aMay_1        ; "May"
                dq offset aJune_1       ; "June"
                dq offset aJuly_1       ; "July"
                dq offset aAugust_1     ; "August"
                dq offset aSeptember_1  ; "September"
                dq offset aOctober_1    ; "October"
                dq offset aNovember_1   ; "November"
                dq offset aDecember_1   ; "December"
aJanuary_1      db 'January',0          ; DATA XREF: sub_140001020+4
                                        ; .data:off_140011000
aFebruary_1     db 'February',0         ; DATA XREF: .data:0000000140011008
                align 4
aMarch_1        db 'March',0            ; DATA XREF: .data:0000000140011010
                align 4
aApril_1        db 'April',0            ; DATA XREF: .data:0000000140011018
```

Les noms des mois se trouvent juste après.

Notre programme est minuscule, il n'y a donc pas beaucoup de données à mettre dans le segment de données, juste les noms des mois. Mais il faut noter qu'il peut y avoir ici vraiment *n'importe quoi* que l'éditeur de lien aurait décidé d'y mettre.

Donc, que se passe-t-il si nous passons 12 à la fonction? Le 13ème élément va être renvoyé.

Voyons comment le CPU traite les octets en une valeur 64-bit:

Listing 1.243 : Fichier exécutable dans IDA

```
off_140011000   dq offset qword_140011060
                                        ; DATA XREF: .text:0000000140001003
                dq offset aFebruary_1   ; "February"
                dq offset aMarch_1      ; "March"
                dq offset aApril_1      ; "April"
                dq offset aMay_1        ; "May"
                dq offset aJune_1       ; "June"
                dq offset aJuly_1       ; "July"
                dq offset aAugust_1     ; "August"
                dq offset aSeptember_1  ; "September"
                dq offset aOctober_1    ; "October"
                dq offset aNovember_1   ; "November"
                dq offset aDecember_1   ; "December"
qword_140011060 dq 797261756E614Ah      ; DATA XREF: sub_140001020+4
                                        ; .data:off_140011000
aFebruary_1     db 'February',0         ; DATA XREF: .data:0000000140011008
```

```
                align 4
aMarch_1        db 'March',0            ; DATA XREF: .data:0000000140011010
```

Et c'est 0x797261756E614A.

Peu après, une autre fonction (supposons, une qui traite des chaînes) pourrait essayer de lire des octets à cette adresse, y attendant une chaîne-C.

Plus probablement, ça planterait, car cette valeur ne ressemble pas à une adresse valide.

Protection contre les débordements de tampon

> Si quelque chose peut mal tourner, ça tournera mal
>
> Loi de Murphy

Il est un peu naïf de s'attendre à ce que chaque programmeur qui utilisera votre fonction ou votre bibliothèque ne passera jamais un argument plus grand que 11.

Il existe une philosophie qui dit «échouer tôt et échouer bruyamment» ou «échouer rapidement», qui enseigne de remonter les problèmes le plus tôt possible et d'arrêter.

Une telle méthode en C/C++ est les assertions.

Nous pouvons modifier notre programme pour qu'il échoue si une valeur incorrecte est passée:

Listing 1.244 : assert() ajoutée

```
const char* get_month1_checked (int month)
{
        assert (month<12);
        return month1[month];
};
```

La macro assertion vérifie que la validité des valeurs à chaque démarrage de fonction et échoue si l'expression est fausse.

Listing 1.245 : MSVC 2013 x64 avec optimisation

```
$SG3143 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '.', 00H
        DB      'c', 00H, 00H, 00H
$SG3144 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '<', 00H
        DB      '1', 00H, '2', 00H, 00H, 00H

month$ = 48
get_month1_checked PROC
$LN5:
        push    rbx
        sub     rsp, 32
        movsxd  rbx, ecx
```

```
        cmp     ebx, 12
        jl      SHORT $LN3@get_month1
        lea     rdx, OFFSET FLAT:$SG3143
        lea     rcx, OFFSET FLAT:$SG3144
        mov     r8d, 29
        call    _wassert
$LN3@get_month1:
        lea     rcx, OFFSET FLAT:month1
        mov     rax, QWORD PTR [rcx+rbx*8]
        add     rsp, 32
        pop     rbx
        ret     0
get_month1_checked ENDP
```

En fait, assert() n'est pas une fonction, mais une macro. Elle teste une condition, puis passe le numéro de ligne et le nom du fichier à une autre fonction qui rapporte cette information à l'utilisateur.

Ici nous voyons qu'à la fois le nom du fichier et la condition sont encodés en UTF-16. Le numéro de ligne est aussi passé (c'est 29).

Le mécanisme est sans doute le même dans tous les compilateurs. Voici ce que fait GCC:

Listing 1.246 : GCC 4.9 x64 avec optimisation

```
.LC1:
        .string "month.c"
.LC2:
        .string "month<12"

get_month1_checked:
        cmp     edi, 11
        jg      .L6
        movsx   rdi, edi
        mov     rax, QWORD PTR month1[0+rdi*8]
        ret
.L6:
        push    rax
        mov     ecx, OFFSET FLAT:__PRETTY_FUNCTION__.2423
        mov     edx, 29
        mov     esi, OFFSET FLAT:.LC1
        mov     edi, OFFSET FLAT:.LC2
        call    __assert_fail

__PRETTY_FUNCTION__.2423:
        .string "get_month1_checked"
```

Donc la macro dans GCC passe aussi le nom de la fonction par commodité.

Rien n'est vraiment gratuit, et c'est également vrai pour les tests de validité.

Ils rendent votre programme plus lent, en particulier si la macro assert() est utilisée dans des petites fonctions à durée critique.

Donc MSCV, par exemple, laisse les tests dans les compilations debug, mais ils disparaissent dans celles de release.

Les noyaux de Microsoft Windows NT existent en versions «checked» et «free». [133].

Le premier a des tests de validation (d'où, «checked»), le second n'en a pas (d'où, «free/libre» de tests).

Bien sûr, le noyau «checked» fonctionne plus lentement à cause de ces tests, donc il n'est utilisé que pour des sessions de debug.

Accéder à un caractère spécifique

Un tableau de pointeurs sur des chaînes peut être accédé comme ceci[134] :

```
#include <stdio.h>

const char* month[]=
{
        "janvier", "fevrier", "mars", "avril",
        "mai", "juin", "juillet", "aout",
        "septembre", "octobre", "novembre", "decembre"
};

int main()
{
        // 4ème mois, 5ème caractère:
        printf ("%c\n", month[3][4]);
};
```

…puisque l'expression *month[3]* a un type *const char**. Et donc, le 5ème caractère est extrait de cette expression en ajoutant 4 octets à cette adresse.

À propos, la liste d'arguments passée à la fonction *main()* a le même type de données:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        printf ("3ème argument, 2ème caractère: %c\n", argv[3][1]);
};
```

Il est très important de comprendre, que, malgré la syntaxe similaire, c'est différent d'un tableau à deux dimensions, dont nous allons parler plus tard.

Une autre chose importante à noter: les chaînes considérées doivent être encodées dans un système où chaque caractère occupe un seul octet, comme l'ASCII[135] ou l'ASCII étendu. UTF-8 ne fonctionnera pas ici.

[133] msdn.microsoft.com/en-us/library/windows/hardware/ff543450(v=vs.85).aspx
[134] Lisez l'avertissement dans la NDT ici 1.26.5 on page 364
[135] American Standard Code for Information Interchange

1.26.6 Tableaux multidimensionnels

En interne, un tableau multidimensionnel est pratiquement la même chose qu'un tableau linéaire.

Puisque la mémoire d'un ordinateur est linéaire, c'est un tableau uni-dimensionnel. Par commodité, ce tableau multidimensionnel peut facilement être représenté comme un uni-dimensionnel.

Par exemple, voici comment les éléments du tableau 3*4 sont placés dans un tableau uni-dimensionnel de 12 éléments:

Offset en mémoire	élément du tableau
0	[0][0]
1	[0][1]
2	[0][2]
3	[0][3]
4	[1][0]
5	[1][1]
6	[1][2]
7	[1][3]
8	[2][0]
9	[2][1]
10	[2][2]
11	[2][3]

Tab. 1.3: Tableau en deux dimensions représenté en mémoire en une dimension

Voici comment chacun des éléments du tableau 3*4 sont placés en mémoire:

0	1	2	3
4	5	6	7
8	9	10	11

Tab. 1.4: Adresse mémoire de chaque élément d'un tableau à deux dimensions

Donc, afin de calculer l'adresse de l'élément voulu, nous devons d'abord multiplier le premier index par 4 (largeur du tableau) et puis ajouter le second index. Ceci est appelé *row-major order* (ordre ligne d'abord), et c'est la méthode de représentation des tableaux et des matrices au moins en C/C++ et Python. Le terme *row-major order* est de l'anglais signifiant: « d'abord, écrire les éléments de la première ligne, puis ceux de la seconde ligne …et enfin les éléments de la dernière ligne ».

Une autre méthode de représentation est appelée *column-major order* (ordre colonne d'abord) (les indices du tableau sont utilisés dans l'ordre inverse) et est utilisé au moins en ForTran, MATLAB et R. Le terme *column-major order* est de l'anglais signifiant: « d'abord, écrire les éléments de la première colonne, puis ceux de la seconde colonne …et enfin les éléments de la dernière colonne ».

Quelle méthode est la meilleure?

En général, en termes de performance et de mémoire cache, le meilleur schéma pour l'organisation des données est celui dans lequel les éléments sont accédés séquentiellement.

Donc si votre fonction accède les données par ligne, *row-major order* est meilleur, et vice-versa.

Exemple de tableau à 2 dimensions

Nous allons travailler avec un tableau de type *char*, qui implique que chaque élément n'a besoin que d'un octet en mémoire.

Exemple de remplissage d'une ligne

Remplissons la seconde ligne avec les valeurs 0..3:

Listing 1.247 : Exemple de remplissage d'une ligne

```
#include <stdio.h>

char a[3][4];

int main()
{
        int x, y;

        // effacer le tableau
        for (x=0; x<3; x++)
                for (y=0; y<4; y++)
                        a[x][y]=0;

        // remplir la 2ème ligne avec 0..3
        for (y=0; y<4; y++)
                a[1][y]=y;
};
```

Les trois lignes sont entourées en rouge. Nous voyons que la seconde ligne a maintenant les valeurs 0, 1, 2 et 3:

```
Address   Hex dump
00C33370  00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00
00C33380  02 00 00 00 C3 66 47 4E C3 66 47 4E 00 00 00 00
00C33390  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Fig. 1.92: OllyDbg : le tableau est rempli

Exemple de remplissage d'une colonne

Remplissons la troisième colonne avec les valeurs: 0..2:

Listing 1.248 : Exemple de remplissage d'une colonne

```
#include <stdio.h>

char a[3][4];

int main()
{
        int x, y;

        // effacer le tableau
        for (x=0; x<3; x++)
                for (y=0; y<4; y++)
                        a[x][y]=0;

        // remplir la troisième colonne avec 0..2:
        for (x=0; x<3; x++)
                a[x][2]=x;
};
```

Les trois lignes sont entourées en rouge ici.

Nous voyons que dans chaque ligne, à la troisième position, ces valeurs sont écrites: 0, 1 et 2.

```
Address  Hex dump
01033380 00 00 00 00 00 00 01 00 00 00 02 00 02 00 00 00
01033390 00 00 00 00 1E AA EF 31 1E AA EF 31 00 00 00 00
010333A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
010333B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Fig. 1.93: OllyDbg : le tableau est rempli

Accéder à un tableau en deux dimensions comme un à une dimension

Nous pouvons facilement nous assurer qu'il est possible d'accéder à un tableau en deux dimensions d'au moins deux façons:

```
#include <stdio.h>

char a[3][4];

char get_by_coordinates1 (char array[3][4], int a, int b)
{
        return array[a][b];
};

char get_by_coordinates2 (char *array, int a, int b)
{
        // traiter le tableau en entrée comme uni-dimensionnel
        // 4 est ici la largeur du tableau
        return array[a*4+b];
};

char get_by_coordinates3 (char *array, int a, int b)
```

```
{
        // traiter le tableau en entrée comme un pointeur
        // calculer l'adresse, y prendre une valeur
        // 4 est ici la largeur du tableau
        return *(array+a*4+b);
};

int main()
{
        a[2][3]=123;
        printf ("%d\n", get_by_coordinates1(a, 2, 3));
        printf ("%d\n", get_by_coordinates2(a, 2, 3));
        printf ("%d\n", get_by_coordinates3(a, 2, 3));
};
```

Compilons[136] le et lançons le: il montre des valeurs correctes.

Ce que MSVC 2013 a généré est fascinant, les trois routines sont les mêmes!

Listing 1.249 : MSVC 2013 avec optimisation x64

```
array$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
; RCX=adresse du tableau
; RDX=a
; R8=b
        movsxd  rax, r8d
; EAX=b
        movsxd  r9, edx
; R9=a
        add     rax, rcx
; RAX=b+adresse du tableau
        movzx   eax, BYTE PTR [rax+r9*4]
; AL=charger l'octet à l'adresse RAX+R9*4=b+adresse du tableau+a*4=adresse du
    tableau+a*4+b
        ret     0
get_by_coordinates3 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates2 PROC
        movsxd  rax, r8d
        movsxd  r9, edx
        add     rax, rcx
        movzx   eax, BYTE PTR [rax+r9*4]
        ret     0
get_by_coordinates2 ENDP

array$ = 8
```

[136]Ce programme doit être compilé comme un programme C, pas C++, sauvegardez-le dans un fichier avecl'extention .c pour le compiler avec MSVC

```
a$ = 16
b$ = 24
get_by_coordinates1 PROC
        movsxd  rax, r8d
        movsxd  r9, edx
        add     rax, rcx
        movzx   eax, BYTE PTR [rax+r9*4]
        ret     0
get_by_coordinates1 ENDP
```

GCC génère des routines équivalentes, mais légèrement différentes:

Listing 1.250 : GCC 4.9 x64 avec optimisation

```
; RDI=adresse du tableau
; RSI=a
; RDX=b

get_by_coordinates1:
; étendre le signe sur 64-bit des valeurs 32-bit en entrée "a" et "b"
        movsx   rsi, esi
        movsx   rdx, edx
        lea     rax, [rdi+rsi*4]
; RAX=RDI+RSI*4=adresse du tableau+a*4
        movzx   eax, BYTE PTR [rax+rdx]
; AL=charger l'octet à l'adresse RAX+RDX=adresse du tableau+a*4+b
        ret

get_by_coordinates2:
        lea     eax, [rdx+rsi*4]
; RAX=RDX+RSI*4=b+a*4
        cdqe
        movzx   eax, BYTE PTR [rdi+rax]
; AL=charger l'octet à l'adresse RDI+RAX=adresse du tableau+b+a*4
        ret

get_by_coordinates3:
        sal     esi, 2
; ESI=a<<2=a*4
; étendre le signe sur 64-bit des valeurs 32-bit en entrée "a*4" et "b"
        movsx   rdx, edx
        movsx   rsi, esi
        add     rdi, rsi
; RDI=RDI+RSI=adresse du tableau+a*4
        movzx   eax, BYTE PTR [rdi+rdx]
; AL=charger l'octet à l'adresse RDI+RAX=adresse du tableau+a*4+b
        ret
```

Exemple de tableau à trois dimensions

C’est la même chose pour des tableaux multidimensionnels.

Nous allons travailler avec des tableaux de type *int* : chaque élément nécessite 4 octets en mémoire.

Voyons ceci:

Listing 1.251 : simple exemple

```
#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
        a[x][y][z]=value;
};
```

x86

Nous obtenons (MSVC 2010) :

Listing 1.252 : MSVC 2010

```
_DATA    SEGMENT
COMM     _a:DWORD:01770H
_DATA    ENDS
PUBLIC   _insert
_TEXT    SEGMENT
_x$ = 8                 ; taille = 4
_y$ = 12                ; taille = 4
_z$ = 16                ; taille = 4
_value$ = 20            ; taille = 4
_insert    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _x$[ebp]
    imul   eax, 2400                    ; eax=600*4*x
    mov    ecx, DWORD PTR _y$[ebp]
    imul   ecx, 120                     ; ecx=30*4*y
    lea    edx, DWORD PTR _a[eax+ecx]   ; edx=a + 600*4*x + 30*4*y
    mov    eax, DWORD PTR _z$[ebp]
    mov    ecx, DWORD PTR _value$[ebp]
    mov    DWORD PTR [edx+eax*4], ecx   ; *(edx+z*4)=valeur
    pop    ebp
    ret    0
_insert    ENDP
_TEXT      ENDS
```

Rien de particulier. Pour le calcul de l'index, trois arguments en entrée sont utilisés dans la formule $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$, pour représenter le tableau comme multidimensionnel. N'oubliez pas que le type *int* est 32-bit (4 octets), donc tous les coefficients doivent être multipliés par 4.

Listing 1.253 : GCC 4.4.1

```
        public insert
insert  proc near

x       = dword ptr  8
y       = dword ptr  0Ch
z       = dword ptr  10h
value   = dword ptr  14h

        push    ebp
        mov     ebp, esp
        push    ebx
        mov     ebx, [ebp+x]
        mov     eax, [ebp+y]
        mov     ecx, [ebp+z]
        lea     edx, [eax+eax]    ; edx=y*2
        mov     eax, edx          ; eax=y*2
        shl     eax, 4            ; eax=(y*2)<<4 = y*2*16 = y*32
        sub     eax, edx          ; eax=y*32 - y*2=y*30
        imul    edx, ebx, 600     ; edx=x*600
        add     eax, edx          ; eax=eax+edx=y*30 + x*600
        lea     edx, [eax+ecx]    ; edx=y*30 + x*600 + z
        mov     eax, [ebp+value]
        mov     dword ptr ds:a[edx*4], eax  ; *(a+edx*4)=valeur
        pop     ebx
        pop     ebp
        retn
insert  endp
```

Le compilateur GCC fait cela différemment.

Pour une des opérations du calcul ($30y$), GCC produit un code sans instruction de multiplication. Voici comment il fait: $(y + y) \ll 4 - (y + y) = (2y) \ll 4 - 2y = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y$. Ainsi, pour le calcul de $30y$, seulement une addition, un décalage de bit et une soustraction sont utilisés. Ceci fonctionne plus vite.

ARM + sans optimisation Xcode 4.6.3 (LLVM) (Mode Thumb)

Listing 1.254 : sans optimisation Xcode 4.6.3 (LLVM) (Mode Thumb)

```
_insert

value   = -0x10
z       = -0xC
y       = -8
x       = -4

; allouer de l'espace sur la pile locale pour 4 valeurs de type int
SUB     SP, SP, #0x10
MOV     R9, 0xFC2 ; a
ADD     R9, PC
LDR.W   R9, [R9] ; prendre le pointeur sur le tableau
```

```
STR     R0, [SP,#0x10+x]
STR     R1, [SP,#0x10+y]
STR     R2, [SP,#0x10+z]
STR     R3, [SP,#0x10+value]
LDR     R0, [SP,#0x10+value]
LDR     R1, [SP,#0x10+z]
LDR     R2, [SP,#0x10+y]
LDR     R3, [SP,#0x10+x]
MOV     R12, 2400
MUL.W   R3, R3, R12
ADD     R3, R9
MOV     R9, 120
MUL.W   R2, R2, R9
ADD     R2, R3
LSLS    R1, R1, #2 ; R1=R1<<2
ADD     R1, R2
STR     R0, [R1]   ; R1 - adresse de l'élément du tableau
; libérer le chunk sur la pile locale, alloué pour 4 valeurs de type int
ADD     SP, SP, #0x10
BX      LR
```

LLVM sans optimisation sauve toutes les variables dans la pile locale, ce qui est redondant.

L'adresse de l'élément du tableau est calculée par la formule vue précédemment.

ARM + avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb)

Listing 1.255 : avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb)

```
_insert
MOVW    R9, #0x10FC
MOV.W   R12, #2400
MOVT.W  R9, #0
RSB.W   R1, R1, R1,LSL#4  ; R1 - y. R1=y<<4 - y = y*16 - y = y*15
ADD     R9, PC
LDR.W   R9, [R9]          ; R9 = pointeur sur la tableau a
MLA.W   R0, R0, R12, R9   ; R0 - x, R12 - 2400, R9 - pointeur sur a.
    R0=x*2400 + ptr sur a
ADD.W   R0, R0, R1,LSL#3  ; R0 = R0+R1<<3 = R0+R1*8 = x*2400 + ptr sur a +
    y*15*8 =
                          ; ptr sur a + y*30*4 + x*600*4
STR.W   R3, [R0,R2,LSL#2] ; R2 - z, R3 - valeur. adresse=R0+z*4 =
                          ; ptr sur a + y*30*4 + x*600*4 + z*4
BX      LR
```

L'astuce de remplacer la multiplication par des décalage, addition et soustraction que nous avons déjà vue est aussi utilisée ici.

Ici, nous voyons aussi une nouvelle instruction: RSB (*Reverse Subtract*).

Elle fonctionne comme SUB, mais échange ses opérandes l'un avec l'autre avant l'exécution. Pourquoi? SUB et RSB sont des instructions auxquelles un coefficient de

décalage peut être appliqué au second opérande: (LSL#4).

Mais ce coefficient ne peut être appliqué qu'au second opérande.

C'est bien pour des opérations commutatives comme l'addition ou la multiplication (les opérandes peuvent être échangés sans changer le résultat).

Mais la soustraction est une opération non commutative, donc RSB existe pour ces cas.

MIPS

Mon exemple est minuscule, donc le compilateur GCC a décidé de mettre le tableau a dans la zone de 64KiB adressable par le Global Pointer.

Listing 1.256 : GCC 4.4.5 avec optimisation (IDA)

```
insert:
; $a0=x
; $a1=y
; $a2=z
; $a3=valeur
                sll     $v0, $a0, 5
; $v0 = $a0<<5 = x*32
                sll     $a0, 3
; $a0 = $a0<<3 = x*8
                addu    $a0, $v0
; $a0 = $a0+$v0 = x*8+x*32 = x*40
                sll     $v1, $a1, 5
; $v1 = $a1<<5 = y*32
                sll     $v0, $a0, 4
; $v0 = $a0<<4 = x*40*16 = x*640
                sll     $a1, 1
; $a1 = $a1<<1 = y*2
                subu    $a1, $v1, $a1
; $a1 = $v1-$a1 = y*32-y*2 = y*30
                subu    $a0, $v0, $a0
; $a0 = $v0-$a0 = x*640-x*40 = x*600
                la      $gp, __gnu_local_gp
                addu    $a0, $a1, $a0
; $a0 = $a1+$a0 = y*30+x*600
                addu    $a0, $a2
; $a0 = $a0+$a2 = y*30+x*600+z
; charger l'adresse de la table:
                lw      $v0, (a & 0xFFFF)($gp)
; multiplier l'index par 4 pour avancer d'un élément du tableau:
                sll     $a0, 2
; ajouter l'index multiplié et l'adresse de la table:
                addu    $a0, $v0, $a0
; stocker la valeur dans la table et retourner:
                jr      $ra
                sw      $a3, 0($a0)

                .comm a:0x1770
```

Obtenir la dimension d'un tableau multidimensionnel

Toute fonction de traitement de chaîne, à laquelle un tableau de caractère lui est passée, ne peut pas en déduire la taille de ce tableau en entrée.

Par exemple:

```
int get_element(int array[10][20], int x, int y)
{
        return array[x][y];
};

int main()
{
        int array[10][20];

        get_element(array, 4, 5);
};
```

…si compilé (par n'importe quel compilateur) et ensuite décompilé par Hex-Rays:

```
int get_element(int *array, int x, int y)
{
  return array[20 * x + y];
}
```

Il n'y a pas moyen de trouver la taille de la première dimension. Si la valeur x passée est trop grosse, un dépassement de tampon peut se produire, un élément d'un endroit aléatoire en mémoire sera lu.

Et un tableau 3D:

```
int get_element(int array[10][20][30], int x, int y, int z)
{
        return array[x][y][z];
};

int main()
{
        int array[10][20][30];

        get_element(array, 4, 5, 6);
};
```

Hex-Rays:

```
int get_element(int *array, int x, int y, int z)
{
  return array[600 * x + z + 30 * y];
}
```

À nouveau, seules deux des 3 dimensions peuvent être déduites.

Plus d'exemples

L'écran de l'ordinateur est représenté comme un tableau 2D, mais le buffer vidéo est un tableau linéaire 1D. Nous en parlons ici: 8.15.2 on page 1184.

Un autre exemple dans ce livre est le jeu Minesweeper: son champ est aussi un tableau à deux dimensions: 8.4 on page 1047.

1.26.7 Ensemble de chaînes comme un tableau à deux dimensions

Retravaillons la fonction qui renvoie le nom d'un mois: listado.1.235.

Comme vous le voyez, au moins une opération de chargement en mémoire est nécessaire pour préparer le pointeur sur la chaîne représentant le nom du mois.

Est-il possible de se passer de cette opération de chargement en mémoire?

En fait oui, si vous représentez la liste de chaînes comme un tableau à deux dimensions:

```
#include <stdio.h>
#include <assert.h>

const char month2[12][10]=
{
        { 'j','a','n','v','i','e','r',  0,  0,  0 },
        { 'f','e','b','v','r','i','e','r',  0,  0 },
        { 'm','a','s',  0,  0,  0,  0,  0,  0,  0 },
        { 'a','v','r','i','l',  0,  0,  0,  0,  0 },
        { 'm','a','i',  0,  0,  0,  0,  0,  0,  0 },
        { 'j','u','i','n',  0,  0,  0,  0,  0,  0 },
        { 'j','u','i','l','l','e','t',  0,  0,  0 },
        { 'a','o','u','t',  0,  0,  0,  0,  0,  0 },
        { 's','e','p','t','e','m','b','r','e',  0 },
        { 'o','c','t','o','b','r','e',  0,  0,  0 },
        { 'n','o','v','e','m','b','r','e',  0,  0 },
        { 'd','e','c','e','m','b','r','e',  0,  0 }
};

// dans l'intervalle 0..11
const char* get_month2 (int month)
{
        return &month2[month][0];
};
```

Voici ce que nous obtenons:

Listing 1.257 : MSVC 2013 x64 avec optimisation

```
month2  DB      04aH
        DB      061H
        DB      06eH
        DB      075H
```

```
        DB      061H
        DB      072H
        DB      079H
        DB      00H
        DB      00H
        DB      00H
...

get_month2 PROC
; étendre le signe de l'argument en entrée sur 64-bit
        movsxd  rax, ecx
        lea     rcx, QWORD PTR [rax+rax*4]
; RCX=mois+mois*4=mois*5
        lea     rax, OFFSET FLAT:month2
; RAX=pointeur sur la table
        lea     rax, QWORD PTR [rax+rcx*2]
; RAX=pointeur sur la table + RCX*2=pointeur sur la table + mois*5*2=pointeur
    sur la table + mois*10
        ret     0
get_month2 ENDP
```

Il n'y a pas du tout d'accès à la mémoire.

Tout ce que fait cette fonction, c'est de calculer le point où le premier caractère du nom du mois se trouve: $pointeur_sur_la_table + mois * 10$.

Il y a deux instructions LEA, qui fonctionnent en fait comme plusieurs instructions MUL et MOV.

La largeur du tableau est de 10 octets.

En effet, la chaîne la plus longue ici—«septembre »—fait 9 octets, plus l'indicateur de fin de chaîne 0, ça fait 10 octets

Le reste du nom de chaque mois est complété par des zéros, afin d'occuper le même espace (10 octets).

Donc, notre fonction fonctionne même plus vite, car toutes les chaînes débutent à une adresse qui peut être facilement calculée.

GCC 4.9 avec optimisation fait encore plus court:

Listing 1.258 : GCC 4.9 x64 avec optimisation

```
        movsx   rdi, edi
        lea     rax, [rdi+rdi*4]
        lea     rax, month2[rax+rax]
        ret
```

LEA est aussi utilisé ici pour la multiplication par 10.

Les compilateurs sans optimisations génèrent la multiplication différemment.

Listing 1.259 : GCC 4.9 x64 sans optimisation

```
get_month2:
        push    rbp
```

```
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        movsx   rdx, eax
; RDX = valeur entrée avec signe étendu
        mov     rax, rdx
; RAX = mois
        sal     rax, 2
; RAX = mois<<2 = mois*4
        add     rax, rdx
; RAX = RAX+RDX = mois*4+mois = mois*5
        add     rax, rax
; RAX = RAX*2 = mois*5*2 = mois*10
        add     rax, OFFSET FLAT:month2
; RAX = mois*10 + pointeur sur la table
        pop     rbp
        ret
```

MSVC sans optimisation utilise simplement l'instruction IMUL :

Listing 1.260 : MSVC 2013 x64 sans optimisation

```
month$ = 8
get_month2 PROC
        mov     DWORD PTR [rsp+8], ecx
        movsxd  rax, DWORD PTR month$[rsp]
; RAX = étendre le signe de la valeur entrée sur 64-bit
        imul    rax, rax, 10
; RAX = RAX*10
        lea     rcx, OFFSET FLAT:month2
; RCX = pointeur sur la table
        add     rcx, rax
; RCX = RCX+RAX = pointeur sur la table+mois*10
        mov     rax, rcx
; RAX = pointeur sur la table+mois*10
        mov     ecx, 1
; RCX = 1
        imul    rcx, rcx, 0
; RCX = 1*0 = 0
        add     rax, rcx
; RAX = pointeur sur la table+mois*10 + 0 = pointeur sur la table+mois*10
        ret     0
get_month2 ENDP
```

Mais une chose est est curieuse: pourquoi ajouter une multiplication par zéro et ajouter zéro au résultat final?

Ceci ressemble à une bizarrerie du générateur de code du compilateur, qui n'a pas été détectée par les tests du compilateur (le code résultant fonctionne correctement après tout). Nous examinons volontairement de tels morceaux de code, afin que le lecteur prenne conscience qu'il ne doit parfois pas se casser la tête sur des artefacts de compilateur.

32-bit ARM

Keil avec optimisation pour le mode Thumb utilise l'instruction de multiplication MULS :

Listing 1.261 : avec optimisation Keil 6/2013 (Mode Thumb)

```
; R0 = mois
        MOVS     r1,#0xa
; R1 = 10
        MULS     r0,r1,r0
; R0 = R1*R0 = 10*mois
        LDR      r1,|L0.68|
; R1 = pointer sur la table
        ADDS     r0,r0,r1
; R0 = R0+R1 = 10*mois + pointer sur la table
        BX       lr
```

Keil avec optimisation pour mode ARM utilise des instructions d'addition et de décalage:

Listing 1.262 : avec optimisation Keil 6/2013 (Mode ARM)

```
; R0 = mois
        LDR      r1,|L0.104|
; R1 = pointeur sur la table
        ADD      r0,r0,r0,LSL #2
; R0 = R0+R0<<2 = R0+R0*4 = mois*5
        ADD      r0,r1,r0,LSL #1
; R0 = R1+R0<<2 = pointeur sur la table + mois*5*2 = pointeur sur la table +
    mois*10
        BX       lr
```

ARM64

Listing 1.263 : GCC 4.9 ARM64 avec optimisation

```
; W0 = mois
        sxtw    x0, w0
; X0 = valeur entrée avec signe étendu
        adrp    x1, .LANCHOR1
        add     x1, x1, :lo12:.LANCHOR1
; X1 = pointeur sur la table
        add     x0, x0, x0, lsl 2
; X0 = X0+X0<<2 = X0+X0*4 = X0*5
        add     x0, x1, x0, lsl 1
; X0 = X1+X0<<1 = X1+X0*2 = pointeur sur la table + X0*10
        ret
```

SXTW est utilisée pour étendre le signe, convertir l'entrée 32-bit en 64-bit et stocker le résultat dans X0.

La paire ADRP/ADD est utilisée pour charger l'adresse de la table.

L'instruction ADD a aussi un suffixe LSL, qui aide avec les multiplications.

MIPS

Listing 1.264 : GCC 4.4.5 avec optimisation (IDA)

```
                .globl get_mois2
get_mois2:
; $a0=mois
                sll     $v0, $a0, 3
; $v0 = $a0<<3 = mois*8
                sll     $a0, 1
; $a0 = $a0<<1 = mois*2
                addu    $a0, $v0
; $a0 = mois*2+mois*8 = mois*10
; charger l'adresse de la table:
                la      $v0, mois2
; ajouter l'adresse de la table et l'index que nous avons calculé et sortir:
                jr      $ra
                addu    $v0, $a0

mois2:         .ascii "janvier"<0>
                .byte 0, 0
aFebruary:      .ascii "fevrier"<0>
                .byte    0
aMarch:         .ascii "mars"<0>
                .byte 0, 0, 0, 0
aApril:         .ascii "avril"<0>
                .byte 0, 0, 0, 0
aMay:           .ascii "mai"<0>
                .byte 0, 0, 0, 0, 0, 0
aJune:          .ascii "juin"<0>
                .byte 0, 0, 0, 0, 0
aJuly:          .ascii "juillet"<0>
                .byte 0, 0, 0, 0, 0
aAugust:        .ascii "aout"<0>
                .byte 0, 0, 0
aSeptember:     .ascii "septembre"<0>
aOctober:       .ascii "octobre"<0>
                .byte 0, 0
aNovember:      .ascii "novembre"<0>
                .byte    0
aDecember:      .ascii "decembre"<0>
                .byte 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Conclusion

C'est une technique surannée de stocker des chaînes de texte. Vous pouvez en trouver beaucoup dans Oracle RDBMS, par exemple. Il est difficile de dire si ça vaut la peine de le faire sur des ordinateurs modernes. Néanmoins, c'est un bon exemple de tableaux, donc il a été ajouté à ce livre.

1.26.8 Conclusion

Un tableau est un ensemble de données adjacentes en mémoire.

C'est vrai pour tout type d'élément, structures incluses.

Pour accéder à un élément spécifique d'un tableau, il suffit de calculer son adresse.

Donc, un pointeur sur un tableau et l'adresse de son premier élément—sont la même chose. C'est pourquoi les expressions `ptr[0]` et `*ptr` sont équivalentes en C/C++. Il est intéressant de noter que Hex-Rays remplace souvent la première par la seconde. Il procède ainsi lorsqu'il n'a aucune idée qu'il travaille avec un pointeur sur le tableau complet et pense que c'est un pointeur sur une seule variable.

1.26.9 Exercices

- http://challenges.re/62
- http://challenges.re/63
- http://challenges.re/64
- http://challenges.re/65
- http://challenges.re/66

1.27 Exemple: un bogue dans Angband

Un ancien jeu rogue-like des années 90 [137] avait un bogue dans l'esprit de "Roadside Picnic"[138] par les frères Strugatsky ou "The Lost Room", une série TV[139] :

> The frog-knows version was abundant of bugs. The funniest of them led to a cunning technique of cheating the game, that was called "mushroom farming". If there were more than a certain number (about five hundred) of objects in the labyrinth, the game would break, and many old things turned into objects thrown to the floor. Accordingly, the player went into the maze, he made such longitudinal grooves there (with a special spell), and walked along the grooves, creating mushrooms with another special spell. When there were a lot of mushrooms, the player put and took, put and took some useful item, and mushrooms one by one turned into this subject. After that, the player returned with hundreds of copies of the useful item.

(Misha "tiphareth" Verbitsky, http://imperium.lenin.ru/CEBEP/arc/3/lightmusic/light.htm)

Et d'autres informations provenant de usenet:

---

137 https://en.wikipedia.org/wiki/Angband_(video_game), http://rephial.org/
138 https://en.wikipedia.org/wiki/Roadside_Picnic
139 https://en.wikipedia.org/wiki/The_Lost_Room

```
From: be...@uswest.com (George Bell)
Subject: [Angband] Multiple artifact copies found (bug?)
Date: Fri, 23 Jul 1993 15:55:08 GMT

Up to 2000 ft I found only 4 artifacts, now my house is littered with the
suckers (FYI, most I've gotten from killing nasties, like Dracoliches and ⤦
    ↳ the
like).  Something really weird is happening now, as I found multiple
copies of the same artifact!  My half-elf ranger is down at 2400 ft on one
level which is particularly nasty.  There is a graveyard plus monsters
surrounded by permanent rock and 2 or 3 other special monster rooms!  I did
so much slashing with my favorite weapon, Crisdurian, that I filled several
rooms nearly to the brim with treasure (as usual, mostly junk).

Then, when I found a way into the big vault, I noticed some of the treasure
had already been identified (in fact it looked strangely familiar!).  Then ⤦
    ↳ I
found *two* Short Swords named Sting (1d6) (+7,+8), and I just ran across a
third copy!  I have seen multiple copies of Gurthang on this level as well.
Is there some limit on the number of items per level which I have exceeded?
This sounds reasonable as all multiple copies I have seen come from this ⤦
    ↳ level.

I'm playing PC angband.  Anybody else had this problem?

-George Bell

Help!  I need a Rod of Restore Life Levels, if there is such a thing.  ⤦
    ↳ These
Graveyards are nasty (Black Reavers and some speed 2 wraith in particular).
```

(https://groups.google.com/forum/#!original/rec.games.moria/jItmfrdGyL8/8csctQqA7PQJ)

```
From: Ceri <cm...@andrew.cmu.edu>
Subject: Re: [Angband] Multiple artifact copies found (bug?)
Date: Fri, 23 Jul 1993 23:32:20 -0400

welcome to the mush bug. if there are more than 256 items
on the floor, things start duplicating. learn to harness
this power and you will win shortly  :>

--Rick
```

(google groups)

```
From: nwe...@soda.berkeley.edu (Nicholas C. Weaver)
Subject: Re: [Angband] Multiple artifact copies found (bug?)
Date: 24 Jul 1993 18:18:05 GMT

In article <74348474...@unix1.andrew.cmu.edu> Ceri <cm...@andrew.cmu.edu> ⤦
    ↳ writes:
>welcome to the mush bug. if there are more than 256 items
```

```
>on the floor, things start duplicating. learn to harness
>this power and you will win shortly  :>
>
>--Rick


        QUestion on this.  Is it only the first 256 items which get
duplicated?  What about the original items?  Etc ETc ETc...

        Oh, for those who like to know about bugs, though, the -n option
(start new character) has the following behavior:

        (this is in version 2.4.Frog.knows on unix)

        If you hit controll-p, you keep your old stats.

        YOu loose all record of artifacts founds and named monsters killed.

        YOu loose all items you are carrying (they get turned into error in
objid()s ).

        You loose your gold.

        You KEEP all the stuff in your house.

        If you kill something, and then quaff a potion of restore life
levels, you are back up to where you were before in EXPERIENCE POINTS!!

        Gaining spells will not work right after this, unless you have a
gain int item (for spellcasters) or gain wis item (for priests/palidans), ⤦
    ⤷ in
which case after performing the above, then take the item back on and off,
you will be able to learn spells normally again.


        This can be exploited, if you are a REAL H0ZER (like me), into
getting multiple artifacts early on.  Just get to a level where you can
pound wormtongue into the ground, kill him, go up, drop your stuff in your
house, buy a few potions of restore exp and high value spellbooks with your
leftover gold, angband -n yourself back to what you were before, and repeat
the process.   Yes, you CAN kill wormtongue multiple times.  :)

        This also allows the creation of a human rogue with dunedain ⤦
    ⤷ warrior
starting stats.

        Of course, such practices are evil, vile, and disgusting.  I take ⤦
    ⤷ no
liability for the results of spreading this information.  Yeah, it's ⤦
    ⤷ another
bug to go onto the pile.
--
Nicholas C. Weaver       perpetual ensign guppy      nwe...@soda.berkeley.⤦
```

```
    ↳ edu
 It is a tale, told by an idiot, full of sound and fury, .signifying ↙
    ↳ nothing.
          Since C evolved out of B, and a C+ is close to a B,
        does that mean that C++ is a devolution of the language?
```

(https://groups.google.com/forum/#!original/rec.games.moria/jItmfrdGyL8/FoQeiccewHAJ)

Le fil de discussion complet.

J'ai trouvé la version avec le bogue (2.4 fk) [140], et on peut voir clairement comment les tableaux globaux sont déclarés:

```
/* Number of dungeon objects */
#define MAX_DUNGEON_OBJ  423

...

int16 sorted_objects[MAX_DUNGEON_OBJ];

/* Identified objects flags                                     */
int8u object_ident[OBJECT_IDENT_SIZE];
int16 t_level[MAX_OBJ_LEVEL+1];
inven_type t_list[MAX_TALLOC];
inven_type inventory[INVEN_ARRAY_SIZE];
```

Peut-être que ceci est une raison. La constante `MAX_DUNGEON_OBJ` est trop petite. Peut-être que les auteurs devraient utiliser des listes chaînées ou d'autres structures de données, qui ont une taille illimitée. Mais les tableaux sont plus simples à utiliser.

Un autre exemple de débordement de tampon dans un tableau défini globalement: 3.31 on page 832.

1.28 Manipulation de bits spécifiques

Beaucoup de fonctions définissent leurs arguments comme des flags dans un champ de bits.

Bien sûr, ils pourraient être substitués par un ensemble de variables de type *bool*, mais ce n'est pas frugal.

1.28.1 Test d'un bit spécifique

x86

Exemple avec l'API win32:

```
        HANDLE fh;
```

[140] http://rephial.org/release/2.4.fk, https://yurichev.com/mirrors/angband-2.4.fk.tar

```
        fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, ↙
    ↘ FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

Nous obtenons (MSVC 2010) :

Listing 1.265 : MSVC 2010

```
        push    0
        push    128             ; 00000080H
        push    4
        push    0
        push    1
        push    -1073741824     ; c0000000H
        push    OFFSET $SG78813
        call    DWORD PTR __imp__CreateFileA@28
        mov     DWORD PTR _fh$[ebp], eax
```

Regardons dans WinNT.h:

Listing 1.266 : WinNT.h

```
#define GENERIC_READ                     (0x80000000L)
#define GENERIC_WRITE                    (0x40000000L)
#define GENERIC_EXECUTE                  (0x20000000L)
#define GENERIC_ALL                      (0x10000000L)
```

Tout est clair, GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000, et c'est la valeur utilisée comme second argument pour la fonction `CreateFile()`[141].

Comment `CreateFile()` va tester ces flags?

Si nous regardons dans KERNEL32.DLL de Windows XP SP3 x86, nous trouverons ce morceau de code dans `CreateFileW` :

Listing 1.267 : KERNEL32.DLL (Windows XP SP3 x86)

```
.text:7C83D429     test    byte ptr [ebp+dwDesiredAccess+3], 40h
.text:7C83D42D     mov     [ebp+var_8], 1
.text:7C83D434     jz      short loc_7C83D417
.text:7C83D436     jmp     loc_7C810817
```

Ici nous voyons l'instruction TEST, toutefois elle n'utilise pas complètement le second argument, mais seulement l'octet le plus significatif et le teste avec le flag `0x40` (ce qui implique le flag `GENERIC_WRITE` ici).

TEST est essentiellement la même chose que `AND`, mais sans sauver le résultat (rappelez vous le cas de `CMP` qui est la même chose que SUB, mais sans sauver le résultat (1.12.4 on page 119)).

La logique de ce bout de code est la suivante:

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

[141]msdn.microsoft.com/en-us/library/aa363858(VS.85).aspx

Si l'instruction AND laisse ce bit, le flag ZF sera mis à zéro et le saut conditionnel JZ ne sera pas effectué. Le saut conditionnel est effectué uniquement su la bit 0x40000000 est absent dans la variable dwDesiredAccess —auquel cas le résultat du AND est 0, ZF est mis à 1 et le saut conditionnel est effectué.

Essayons avec GCC 4.4.1 et Linux:

```
#include <stdio.h>
#include <fcntl.h>

void main()
{
        int handle;

        handle=open ("file", O_RDWR | O_CREAT);
};
```

Nous obtenons:

Listing 1.268 : GCC 4.4.1

```
                public main
main            proc near

var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_1C], 42h
                mov     [esp+20h+var_20], offset aFile ; "file"
                call    _open
                mov     [esp+20h+var_4], eax
                leave
                retn
main            endp
```

Si nous regardons dans la fonction open() de la bibliothèque libc.so.6, c'est seulement un appel système:

Listing 1.269 : open() (libc.so.6)

```
.text:000BE69B     mov     edx, [esp+4+mode] ; mode
.text:000BE69F     mov     ecx, [esp+4+flags] ; flags
.text:000BE6A3     mov     ebx, [esp+4+filename] ; filename
.text:000BE6A7     mov     eax, 5
.text:000BE6AC     int     80h             ; LINUX - sys_open
```

Donc, le champ de bits pour open() est apparemment testé quelque part dans le noyau Linux.

Bien sûr, il est facile de télécharger le code source de la Glibc et du noyau Linux, mais nous voulons comprendre ce qui se passe sans cela.

Donc, à partir de Linux 2.6, lorsque l'appel système sys_open est appelé, le contrôle passe finalement à do_sys_open, et à partir de là—à la fonction do_filp_open() (elle est située ici fs/namei.c dans l'arborescence des sources du noyau).

N.B. Outre le passage des arguments par la pile, il y a aussi une méthode consistant à passer certains d'entre eux par des registres. Ceci est aussi appelé fastcall (6.1.3 on page 955). Ceci fonctionne plus vite puisque le CPU ne doit pas faire d'accès à la pile en mémoire pour lire la valeur des arguments. GCC a l'option *regparm*[142], avec laquelle il est possible de définir le nombre d'arguments qui peuvent être passés par des registres.

Le noyau Linux 2.6 est compilé avec l'option -mregparm=3 [143] [144].

Cela signifie que les 3 premiers arguments sont passés par les registres EAX, EDX et ECX, et le reste via la pile. Bien sûr, si le nombre d'arguments est moins que 3, seule une partie de ces registres seront utilisés.

Donc, téléchargeons le noyau Linux 2.6.31, compilons-le dans Ubuntu: make vmlinux, ouvrons-le dans IDA, et cherchons la fonction do_filp_open(). Au début, nous voyons (les commentaires sont les miens) :

Listing 1.270 : do_filp_open() (noyau Linux kernel 2.6.31)

```
do_filp_open    proc near
...
                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                mov     ebx, ecx
                add     ebx, 1
                sub     esp, 98h
                mov     esi, [ebp+arg_4] ; acc_mode (5ème argument)
                test    bl, 3
                mov     [ebp+var_80], eax ; dfd (1er argument)
                mov     [ebp+var_7C], edx ; pathname (2ème argument)
                mov     [ebp+var_78], ecx ; open_flag (3ème argument)
                jnz     short loc_C01EF684
                mov     ebx, ecx        ; ebx <- open_flag
```

GCC sauve les valeurs des 3 premiers arguments dans la pile locale. Si cela n'était pas fait, le compilateur ne toucherait pas ces registres, et ça serait un environnement trop étroit pour l'allocateur de registres du compilateur.

Cherchons ce morceau de code:

Listing 1.271 : do_filp_open() (noyau Linux 2.6.31)

[142] ohse.de/uwe/articles/gcc-attributes.html#func-regparm
[143] kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f
[144] Voir aussi le fichier arch/x86/include/asm/calling.h dans l'arborescence du noyau

```
loc_C01EF684:       ; CODE XREF: do_filp_open+4F
                test    bl, 40h         ; O_CREAT
                jnz     loc_C01EF810
                mov     edi, ebx
                shr     edi, 11h
                xor     edi, 1
                and     edi, 1
                test    ebx, 10000h
                jz      short loc_C01EF6D3
                or      edi, 2
```

0x40—c'est ce à quoi est égale la macro O_CREAT. Le bit 0x40 de open_flag est testé, et si il est à 1, le saut de l'instruction JNZ suivante est effectué.

ARM

Le bit O_CREAT est testé différemment dans le noyau Linux 3.8.0.

Listing 1.272 : noyau Linux 3.8.0

```
struct file *do_filp_open(int dfd, struct filename *pathname,
                const struct open_flags *op)
{
...
        filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
...
}

static struct file *path_openat(int dfd, struct filename *pathname,
                struct nameidata *nd, const struct open_flags *op, int ↙
    ↘ flags)
{
...
        error = do_last(nd, &path, file, op, &opened, pathname);
...
}


static int do_last(struct nameidata *nd, struct path *path,
                   struct file *file, const struct open_flags *op,
                   int *opened, struct filename *name)
{
...
        if (!(open_flag & O_CREAT)) {
    ...
                error = lookup_fast(nd, path, &inode);
    ...
        } else {
    ...
                error = complete_walk(nd);
        }
...
}
```

Voici à quoi ressemble le noyau compilé pour le mode ARM dans IDA :

Listing 1.273 : do_last() dans vmlinux (IDA)

```
...
.text:C0169EA8      MOV       R9, R3  ; R3 - (4th argument) open_flag
...
.text:C0169ED4      LDR       R6, [R9] ; R6 - open_flag
...
.text:C0169F68      TST       R6, #0x40 ; jumptable C0169F00 default case
.text:C0169F6C      BNE       loc_C016A128
.text:C0169F70      LDR       R2, [R4,#0x10]
.text:C0169F74      ADD       R12, R4, #8
.text:C0169F78      LDR       R3, [R4,#0xC]
.text:C0169F7C      MOV       R0, R4
.text:C0169F80      STR       R12, [R11,#var_50]
.text:C0169F84      LDRB      R3, [R2,R3]
.text:C0169F88      MOV       R2, R8
.text:C0169F8C      CMP       R3, #0
.text:C0169F90      ORRNE     R1, R1, #3
.text:C0169F94      STRNE     R1, [R4,#0x24]
.text:C0169F98      ANDS      R3, R6, #0x200000
.text:C0169F9C      MOV       R1, R12
.text:C0169FA0      LDRNE     R3, [R4,#0x24]
.text:C0169FA4      ANDNE     R3, R3, #1
.text:C0169FA8      EORNE     R3, R3, #1
.text:C0169FAC      STR       R3, [R11,#var_54]
.text:C0169FB0      SUB       R3, R11, #-var_38
.text:C0169FB4      BL        lookup_fast
...
.text:C016A128 loc_C016A128   ; CODE XREF: do_last.isra.14+DC
.text:C016A128      MOV       R0, R4
.text:C016A12C      BL        complete_walk
...
```

TST est analogue à l'instruction TEST en x86. Nous pouvons «pointer» visuellement ce morceau de code grâce au fait que la fonction `lookup_fast()` doit être exécutée dans un cas et `complete_walk()` dans l'autre. Ceci correspond au code source de la fonction `do_last()`. La macro O_CREAT vaut `0x40` ici aussi.

1.28.2 Mettre (à 1) et effacer (à 0) des bits spécifiques

Par exemple:

```
#include <stdio.h>

#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

int f(int a)
{
    int rt=a;
```

```
    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);

    return rt;
};

int main()
{
    f(0x12340678);
};
```

x86

MSVC sans optimisation

Nous obtenons (MSVC 2010) :

Listing 1.274 : MSVC 2010

```
_rt$ = -4         ; size = 4
_a$ = 8           ; size = 4
_f  PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR _rt$[ebp], eax
    mov    ecx, DWORD PTR _rt$[ebp]
    or     ecx, 16384                ; 00004000H
    mov    DWORD PTR _rt$[ebp], ecx
    mov    edx, DWORD PTR _rt$[ebp]
    and    edx, -513                ; fffffdffH
    mov    DWORD PTR _rt$[ebp], edx
    mov    eax, DWORD PTR _rt$[ebp]
    mov    esp, ebp
    pop    ebp
    ret    0
_f  ENDP
```

L'instruction `OR` met un bit à la valeur 1 tout en ignorant les autres bits.

`AND` annule un bit. On peut dire que `AND` copie simplement tous les bits sauf un. En effet, dans le second opérande du `AND` seuls les bits qui doivent être sauvés sont mis (à 1), seul celui qu'on ne veut pas copier ne l'est pas (il est à 0 dans le bitmask). C'est la manière la plus facile de mémoriser la logique.

OllyDbg

Essayons cet exemple dans OllyDbg.

Tout d'abord, regardons la forme binaire de la constante que nous allons utiliser:

`0x200` (0b00000000000000000001000000000) (i.e., le 10ème bit (en comptant depuis le 1er)).

`0x200` inversé est `0xFFFFFDFF` (0b11111111111111111110111111111).

`0x4000` (0b00000000000000100000000000000) (i.e., le 15ème bit).

La valeur d'entrée est: `0x12340678` (0b10010001101000000011001111000). Nous voyons comment elle est chargée:

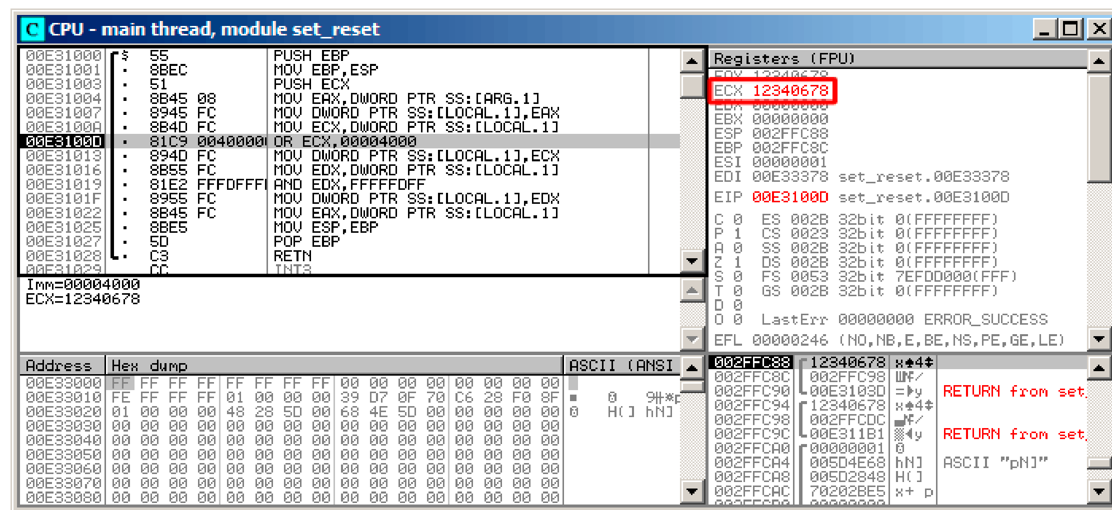


Fig. 1.94: OllyDbg : valeur chargée dans ECX

OR exécuté:

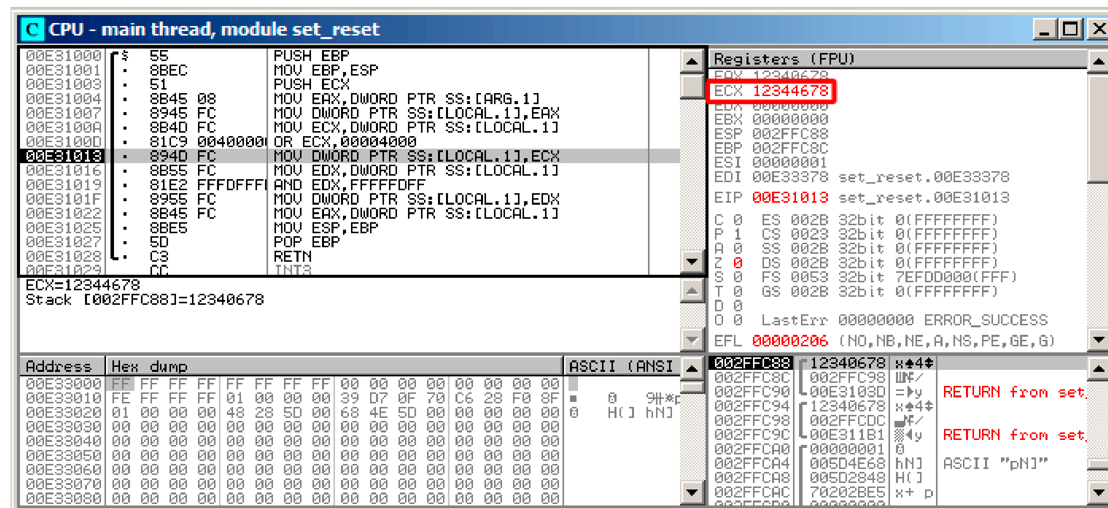


Fig. 1.95: OllyDbg : OR exécuté

Le 15ème bit est mis: `0x12344678` (0b10010001101000100011001111000).

La valeur est encore rechargée (car le compilateur n'est pas en mode avec optimisation) :

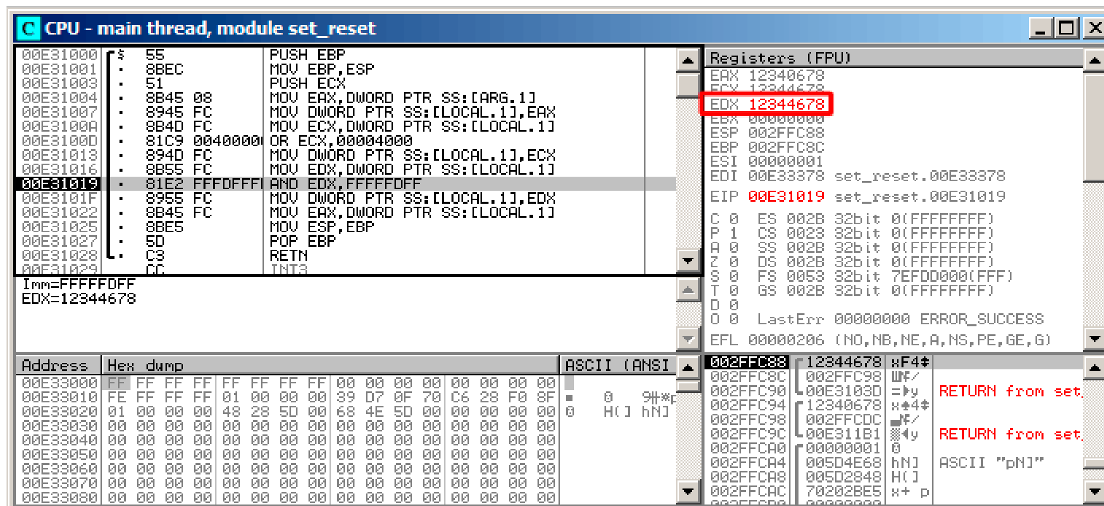


Fig. 1.96: OllyDbg : valeur rechargée dans EDX

AND exécuté:

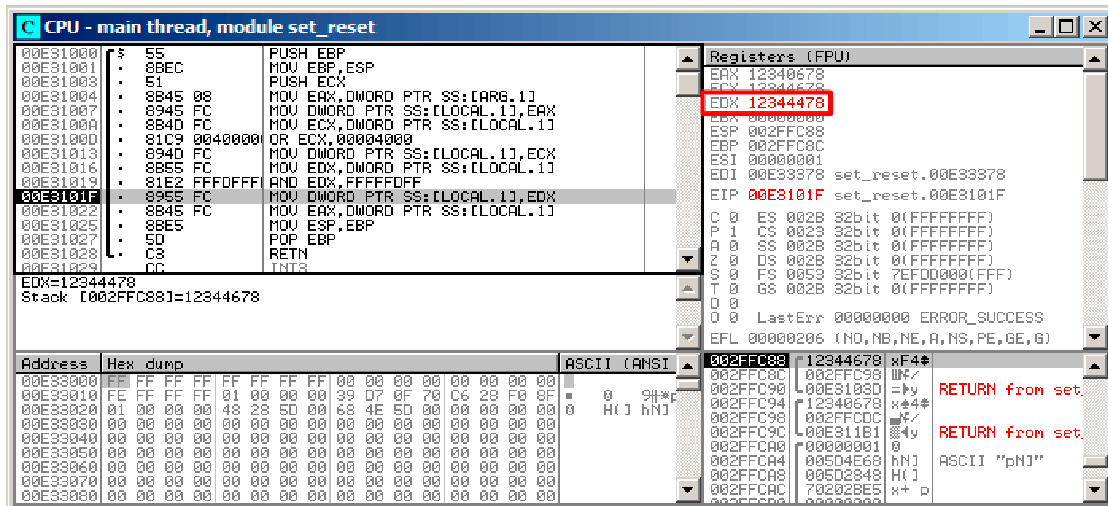


Fig. 1.97: OllyDbg : AND exécuté

Le 10ème bit a été mis à 0 (ou, en d'autres mots, tous les bits ont été laissés sauf le 10ème) et la valeur finale est maintenant 0x12344478 (0b10010001101000100010001111000).

MSVC avec optimisation

Si nous le compilons dans MSVC avec l'option d'optimisation (/Ox), le code est même plus court:

Listing 1.275 : MSVC avec optimisation

```
_a$ = 8                         ; size = 4
_f    PROC
    mov     eax, DWORD PTR _a$[esp-4]
    and     eax, -513           ; fffffdffH
    or      eax, 16384          ; 00004000H
    ret     0
_f    ENDP
```

GCC sans optimisation

Essayons avec GCC 4.4.1 sans optimisation:

Listing 1.276 : GCC sans optimisation

```
                public f
f               proc near
```

```
var_4           = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+var_4], eax
                or      [ebp+var_4], 4000h
                and     [ebp+var_4], 0FFFFFDFFh
                mov     eax, [ebp+var_4]
                leave
                retn
f               endp
```

Il y a du code redondant, toutefois, c'est plus court que la version MSVC sans optimisation.

Maintenant, essayons GCC avec l'option d'optimisation `-O3` :

GCC avec optimisation

Listing 1.277 : GCC avec optimisation

```
                public f
f               proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                pop     ebp
                or      ah, 40h
                and     ah, 0FDh
                retn
f               endp
```

C'est plus court. Il est intéressant de noter que le compilateur travaille avec une partie du registre EAX via le registre AH—qui est la partie du registre EAX située entre les 8ème et 15ème bits inclus.

<table>
<tr><td colspan="8">Octet d'indice</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">RAX^{x64}</td></tr>
<tr><td colspan="4"></td><td colspan="4">EAX</td></tr>
<tr><td colspan="6"></td><td colspan="2">AX</td></tr>
<tr><td colspan="6"></td><td>AH</td><td>AL</td></tr>
</table>

N.B. L'accumulateur du CPU 16-bit 8086 était appelé AX et consistait en deux moitiés de 8-bit—AL (octet bas) et AH (octet haut). Dans le 80386, presque tous les registres

ont été étendus à 32-bit, l'accumulateur a été appelé EAX, mais pour des raisons de compatibilité, ses *anciennes parties* peuvent toujours être accédées par AX/AH/AL.

Puisque tous les CPUs x86 sont des descendants du CPU 16-bit 8086, ces *anciens* opcodes 16-bit sont plus courts que les nouveaux sur 32-bit. C'est pourquoi l'instruction or ah, 40h occupe seulement 3 octets. Il serait plus logique de générer ici or eax, 04000h mais ça fait 5 octets, ou même 6 (dans le cas où le registre du premier opérande n'est pas EAX).

GCC avec optimisation et regparm

Il serait encore plus court en mettant le flag d'optimisation -O3 et aussi regparm=3.

Listing 1.278 : GCC avec optimisation

```
                public f
f               proc near
                push    ebp
                or      ah, 40h
                mov     ebp, esp
                and     ah, 0FDh
                pop     ebp
                retn
f               endp
```

En effet, le premier argument est déjà chargé dans EAX, donc il est possible de travailler avec directement. Il est intéressant de noter qu'à la fois le prologue (push ebp / mov ebp,esp) et l'épilogue (pop ebp) de la fonction peuvent être facilement omis ici, mais sans doute que GCC n'est pas assez bon pour effectuer une telle optimisation de la taille du code. Toutefois, il est préférable que de telles petites fonctions soient des *fonctions inlined* (3.14 on page 649).

ARM + avec optimisation Keil 6/2013 (Mode ARM)

Listing 1.279 : avec optimisation Keil 6/2013 (Mode ARM)

```
02 0C C0 E3          BIC     R0, R0, #0x200
01 09 80 E3          ORR     R0, R0, #0x4000
1E FF 2F E1          BX      LR
```

L'instruction BIC (*BItwise bit Clear*) est une instruction pour mettre à zéro des bits spécifiques. Ceci est comme l'instruction AND, mais avec un opérande inversé. I.e., c'est analogue à la paire d'instructions NOT +AND.

ORR est le «ou logique», analogue à OR en x86.

Jusqu'ici, c'est facile.

ARM + avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.280 : avec optimisation Keil 6/2013 (Mode Thumb)

```
01 21 89 03          MOVS    R1, 0x4000
08 43                ORRS    R0, R1
49 11                ASRS    R1, R1, #5   ; génère 0x200 et le met dans R1
88 43                BICS    R0, R1
70 47                BX      LR
```

Il semble que Keil a décidé que le code en mode Thumb, pour générer `0x200` à partir de `0x4000`, est plus compact que celui pour écrire `0x200` dans un registre arbitraire.

C'est pourquoi, avec l'aide de ASRS (décalage arithmétique vers la droite), cette valeur est calculée comme `0x4000` $\gg 5$.

ARM + avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Listing 1.281 : avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
42 0C C0 E3          BIC             R0, R0, #0x4200
01 09 80 E3          ORR             R0, R0, #0x4000
1E FF 2F E1          BX              LR
```

Le code qui a été généré par LLVM, pourrait être quelque chose comme ça sous la forme de code source:

```
    REMOVE_BIT (rt, 0x4200);
    SET_BIT (rt, 0x4000);
```

Et c'est exactement ce dont nous avons besoin. Mais pourquoi `0x4200` ? Peut-être que c'est un artefact de l'optimiseur de LLVM[145].

Probablement une erreur de l'optimiseur du compilateur, mais le code généré fonctionne malgré tout correctement.

Vous pouvez en savoir plus sur les anomalies de compilateur ici (11.5 on page 1291).

avec optimisation Xcode 4.6.3 (LLVM) pour le mode Thumb génère le même code.

ARM: plus d'informations sur l'instruction `BIC`

Retravaillons légèrement l'exemple:

```
int f(int a)
{
    int rt=a;

    REMOVE_BIT (rt, 0x1234);

    return rt;
};
```

Ensuite Keil 5.03 pour mode ARM avec optimisation fait:

[145] C'était LLVM build 2410.2.00 fourni avec Apple Xcode 4.6.3

```
f PROC
        BIC      r0,r0,#0x1000
        BIC      r0,r0,#0x234
        BX       lr
        ENDP
```

Il y a deux instructions BIC, i.e., les bits 0x1234 sont mis à zéro en deux temps.

C'est parce qu'il n'est pas possible d'encoder 0x1234 dans une instruction BIC, mais il est possible d'encoder 0x1000 et 0x234.

ARM64: GCC (Linaro) 4.9 avec optimisation

GCC en compilant avec optimisation pour ARM64 peut utiliser l'instruction AND au lieu de BIC :

Listing 1.282 : GCC (Linaro) 4.9 avec optimisation

```
f:
        and     w0, w0, -513    ; 0xFFFFFFFFFFFFFDFF
        orr     w0, w0, 16384   ; 0x4000
        ret
```

ARM64: GCC (Linaro) 4.9 sans optimisation

GCC sans optimisation génère plus de code redondant, mais fonctionne comme celui optimisé:

Listing 1.283 : GCC (Linaro) 4.9 sans optimisation

```
f:
        sub     sp, sp, #32
        str     w0, [sp,12]
        ldr     w0, [sp,12]
        str     w0, [sp,28]
        ldr     w0, [sp,28]
        orr     w0, w0, 16384   ; 0x4000
        str     w0, [sp,28]
        ldr     w0, [sp,28]
        and     w0, w0, -513    ; 0xFFFFFFFFFFFFFDFF
        str     w0, [sp,28]
        ldr     w0, [sp,28]
        add     sp, sp, 32
        ret
```

MIPS

Listing 1.284 : GCC 4.4.5 avec optimisation (IDA)

```
f:
; $a0=a
```

```
                ori     $a0, 0x4000
; $a0=a|0x4000
                li      $v0, 0xFFFFFDFF
                jr      $ra
                and     $v0, $a0, $v0
; à la fin: $v0 = $a0 & $v0 = a|0x4000 & 0xFFFFFDFF
```

ORI est, bien sûr, l'opération OR. «I» dans l'instruction signifie que la valeur est intégrée dans le code machine.

Mais après ça, nous avons AND. Il n'y a pas moyen d'utiliser ANDI car il n'est pas possible d'intégrer le nombre 0xFFFFFDFF dans une seule instruction, donc le compilateur doit d'abord charger 0xFFFFFDFF dans le registre $V0 et ensuite génère AND qui prend toutes ses valeurs depuis des registres.

1.28.3 Décalages

Les décalages de bit sont implémentés en C/C++ avec les opérateurs ≪ et ≫. Le x86 ISA possède les instructions SHL (SHift Left / décalage à gauche) et SHR (SHift Right / décalage à droite) pour ceci. Les instructions de décalage sont souvent utilisées pour la division et la multiplication par des puissances de deux: 2^n (e.g., 1, 2, 4, 8, etc.) : 1.24.1 on page 278, 1.24.2 on page 284.

Les opérations de décalage sont aussi si importantes car elles sont souvent utilisées pour isoler des bits spécifiques ou pour construire une valeur à partir de plusieurs bits épars.

1.28.4 Mettre et effacer des bits spécifiques: exemple avec le FPU

Voici comment les bits sont organisés dans le type *float* au format IEEE 754:

(S — signe)

Le signe du nombre est dans le MSB[146]. Est-ce qu'il est possible de changer le signe d'un nombre en virgule flottante sans aucune instruction FPU?

```
#include <stdio.h>

float my_abs (float i)
{
        unsigned int tmp=(*(unsigned int*)&i) & 0x7FFFFFFF;
        return *(float*)&tmp;
};

float set_sign (float i)
```

[146] Bit le plus significatif

```
{
        unsigned int tmp=(*(unsigned int*)&i) | 0x80000000;
        return *(float*)&tmp;
};

float negate (float i)
{
        unsigned int tmp=(*(unsigned int*)&i) ^ 0x80000000;
        return *(float*)&tmp;
};

int main()
{
        printf ("my_abs():\n");
        printf ("%f\n", my_abs (123.456));
        printf ("%f\n", my_abs (-456.123));
        printf ("set_sign():\n");
        printf ("%f\n", set_sign (123.456));
        printf ("%f\n", set_sign (-456.123));
        printf ("negate():\n");
        printf ("%f\n", negate (123.456));
        printf ("%f\n", negate (-456.123));
};
```

Nous avons besoin de cette ruse en C/C++ pour copier vers/depuis des valeurs *float* sans conversion effective. Donc il y a trois fonctions: my_abs() supprime MSB ; `set_sign()` met MSB et negate() l'inverse.

XOR peut être utilisé pour inverser un bit.

x86

Le code est assez simple.

Listing 1.285 : MSVC 2012 avec optimisation

```
_tmp$ = 8
_i$ = 8
_my_abs PROC
        and     DWORD PTR _i$[esp-4], 2147483647 ; 7fffffffH
        fld     DWORD PTR _tmp$[esp-4]
        ret     0
_my_abs ENDP

_tmp$ = 8
_i$ = 8
_set_sign PROC
        or      DWORD PTR _i$[esp-4], -2147483648 ; 80000000H
        fld     DWORD PTR _tmp$[esp-4]
        ret     0
_set_sign ENDP

_tmp$ = 8
_i$ = 8
```

```
_negate PROC
        xor     DWORD PTR _i$[esp-4], -2147483648 ; 80000000H
        fld     DWORD PTR _tmp$[esp-4]
        ret     0
_negate ENDP
```

Une valeur en entrée de type *float* est prise sur la pile, mais traitée comme une valeur entière.

AND et OR supprime et mette le bit désiré. XOR l'inverse.

Enfin, la valeur modifiée est chargée dans ST0,car les nombres en virgule flottante sont renvoyés dans ce registre.

Maintenant essayons l'optimisation de MSVC 2012 pour x64:

Listing 1.286 : MSVC 2012 x64 avec optimisation

```
tmp$ = 8
i$ = 8
my_abs  PROC
        movss   DWORD PTR [rsp+8], xmm0
        mov     eax, DWORD PTR i$[rsp]
        btr     eax, 31
        mov     DWORD PTR tmp$[rsp], eax
        movss   xmm0, DWORD PTR tmp$[rsp]
        ret     0
my_abs  ENDP
_TEXT   ENDS

tmp$ = 8
i$ = 8
set_sign PROC
        movss   DWORD PTR [rsp+8], xmm0
        mov     eax, DWORD PTR i$[rsp]
        bts     eax, 31
        mov     DWORD PTR tmp$[rsp], eax
        movss   xmm0, DWORD PTR tmp$[rsp]
        ret     0
set_sign ENDP

tmp$ = 8
i$ = 8
negate  PROC
        movss   DWORD PTR [rsp+8], xmm0
        mov     eax, DWORD PTR i$[rsp]
        btc     eax, 31
        mov     DWORD PTR tmp$[rsp], eax
        movss   xmm0, DWORD PTR tmp$[rsp]
        ret     0
negate  ENDP
```

La valeur en entrée est passée dans XMM0, puis elle est copiée sur la pile locale et nous voyons des nouvelles instructions: BTR, BTS, BTC.

Ces instructions sont utilisées pour effacer (BTR), mettre (BTS) et inverser (ou faire le complément: BTC) de bits spécifiques. Le bit d'index 31 est le MSB, en comptant depuis 0.

Enfin, le résultat est copié dans XMM0, car les valeurs en virgule flottante sont renvoyées dans XMM0 en environnement Win64.

MIPS

GCC 4.4.5 pour MIPS fait essentiellement la même chose:

Listing 1.287 : GCC 4.4.5 avec optimisation (IDA)

```
my_abs:
; déplacer depuis le coprocesseur 1:
                mfc1    $v1, $f12
                li      $v0, 0x7FFFFFFF
; $v0=0x7FFFFFFF
; faire AND:
                and     $v0, $v1
; déplacer vers le coprocesseur 1:
                mtc1    $v0, $f0
; return
                jr      $ra
                or      $at, $zero ; slot de délai de branchement

set_sign:
; déplacer depuis le coprocesseur 1:
                mfc1    $v0, $f12
                lui     $v1, 0x8000
; $v1=0x80000000
; faire OR:
                or      $v0, $v1, $v0
; déplacer vers le coprocesseur 1:
                mtc1    $v0, $f0
; return
                jr      $ra
                or      $at, $zero ; slot de délai de branchement

negate:
; déplacer depuis le coprocesseur 1:
                mfc1    $v0, $f12
                lui     $v1, 0x8000
; $v1=0x80000000
; do XOR:
                xor     $v0, $v1, $v0
; déplacer vers le coprocesseur 1:
                mtc1    $v0, $f0
; sortir
                jr      $ra
                or      $at, $zero ; slot de délai de branchement
```

Une seule instruction LUI est utilisée pour charger 0x80000000 dans un registre, car

LUI efface les 16 bits bas et ils sont à zéro dans la constante, donc un LUI sans ORI ultérieur est suffisant.

ARM

avec optimisation Keil 6/2013 (Mode ARM)

Listing 1.288 : avec optimisation Keil 6/2013 (Mode ARM)

```
my_abs PROC
; effacer bit:
        BIC      r0,r0,#0x80000000
        BX       lr
        ENDP

set_sign PROC
; faire OR:
        ORR      r0,r0,#0x80000000
        BX       lr
        ENDP

negate PROC
; faire XOR:
        EOR      r0,r0,#0x80000000
        BX       lr
        ENDP
```

Jusqu'ici tout va bien.

ARM a l'instruction BIC, qui efface explicitement un (des) bit(s) spécifique(s). EOR est le nom de l'instruction ARM pour XOR («Exclusive OR / OU exclusif»).

avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.289 : avec optimisation Keil 6/2013 (Mode Thumb)

```
my_abs PROC
        LSLS     r0,r0,#1
; r0=i<<1
        LSRS     r0,r0,#1
; r0=(i<<1)>>1
        BX       lr
        ENDP

set_sign PROC
        MOVS     r1,#1
; r1=1
        LSLS     r1,r1,#31
; r1=1<<31=0x80000000
        ORRS     r0,r0,r1
; r0=r0 | 0x80000000
```

```
        BX       lr
        ENDP

negate PROC
        MOVS     r1,#1
; r1=1
        LSLS     r1,r1,#31
; r1=1<<31=0x80000000
        EORS     r0,r0,r1
; r0=r0 ^ 0x80000000
        BX       lr
        ENDP
```

En ARM, le mode Thumb offre des instructions 16-bit et peu de données peuvent y être encodées, donc ici une paire d'instructions MOVS/LSLS est utilisée pour former la constante 0x80000000. Ça fonctionne comme ceci: $1 << 31 = 0x80000000$.

Le code de my_abs est bizarre et fonctionne pratiquement comme cette expression: $(i << 1) >> 1$. Cette déclaration semble vide de sens. Mais néanmoins, lorsque $input << 1$ est exécuté, le MSB (bit de signe) est simplement supprimé. Puis lorsque la déclaration suivante $result >> 1$ est exécutée, tous les bits sont à nouveau à leur place, mais le MSB vaut zéro, car tous les «nouveaux» bits apparaissant lors d'une opération de décalage sont toujours zéro. C'est ainsi que la paire d'instructions LSLS/LSRS efface le MSB.

avec optimisation GCC 4.6.3 (Raspberry Pi, Mode ARM)

Listing 1.290 : avec optimisation GCC 4.6.3 for Raspberry Pi (Mode ARM)

```
my_abs
; copier depuis S0 vers R2:
                FMRS    R2, S0
; effacer le bit:
                BIC     R3, R2, #0x80000000
; copier depuis R3 vers S0:
                FMSR    S0, R3
                BX      LR

set_sign
; copier depuis S0 vers R2:
                FMRS    R2, S0
; faire OR:
                ORR     R3, R2, #0x80000000
; copier depuis R3 vers S0:
                FMSR    S0, R3
                BX      LR

negate
; copier depuis S0 vers R2:
                FMRS    R2, S0
; faire ADD:
                ADD     R3, R2, #0x80000000
```

```
; copier depuis R3 vers S0:
                FMSR    S0, R3
                BX      LR
```

Lançons Linux pour Raspberry Pi dans QEMU et ça émule un FPU ARM, dons les S-registres sont utilisés pour les nombres en virgule flottante au lieu des R-registres.

L'instruction FMRS copie des données des GPR vers le FPU et retour.

my_abs() et set_sign() ressemblent a ce que l'on attend, mais negate()? Pourquoi est-ce qu'il y a ADD au lieu de XOR ?

C'est dur à croire, mais l'instruction ADD register, 0x80000000 fonctionne tout comme
XOR register, 0x80000000. Tout d'abord, quel est notre but? Le but est de changer le MSB, donc oublions l'opération XOR. Des mathématiques niveau scolaire, nous nous rappelons qu'ajouter une valeur comme 1000 à une autre valeur n'affecte jamais les 3 derniers chiffres. Par exemple: $1234567 + 10000 = 1244567$ (les 4 derniers chiffres ne sont jamais affectés).

Mais ici nous opérons en base décimale et
0x80000000 est 0b10000000000000000000000000000000, i.e., seulement le bit le plus haut est mis.

Ajouter 0x80000000 à n'importe quelle valeur n'affecte jamais les 31 bits les plus bas, mais affecte seulement le MSB. Ajouter 1 à 0 donne 1.

Ajouter 1 à 1 donne 0b10 au format binaire, mais le bit d'indice 32 (en comptant à partir de zéro) est abandonné, car notre registre est large de 32 bit, donc le résultat est 0. C'est pourquoi XOR peut être remplacé par ADD ici.

Il est difficile de dire pourquoi GCC a décidé de faire ça, mais ça fonctionne correctement.

1.28.5 Compter les bits mis à 1

Voici un exemple simple d'une fonction qui compte le nombre de bits mis à 1 dans la valeur en entrée.

Cette opération est aussi appelée «population count »[147].

```
#include <stdio.h>

#define IS_SET(flag, bit)       ((flag) & (bit))

int f(unsigned int a)
{
    int i;
    int rt=0;

    for (i=0; i<32; i++)
        if (IS_SET (a, 1<<i))
            rt++;
```

[147] les CPUs x86 modernes (qui supportent SSE4) ont même une instruction POPCNT pour cela

```
    return rt;
};

int main()
{
    f(0x12345678); // test
};
```

Dans cette boucle, la variable d'itération i prend les valeurs de 0 à 31, donc la déclaration $1 \ll i$ prend les valeurs de 1 à 0x80000000. Pour décrire cette opération en langage naturel, nous dirions *décaler 1 par n bits à gauche*. En d'autres mots, la déclaration $1 \ll i$ produit consécutivement toutes les positions possible pour un bit dans un nombre de 32-bit. Le bit libéré à droite est toujours à 0.

Voici une table de tous les $1 \ll i$ possible for $i = 0 \ldots 31$:

C/C++ expression	Puissance de deux	Forme décimale	Forme hexadécimale
$1 \ll 0$	2^{0}	1	1
$1 \ll 1$	2^{1}	2	2
$1 \ll 2$	2^{2}	4	4
$1 \ll 3$	2^{3}	8	8
$1 \ll 4$	2^{4}	16	0x10
$1 \ll 5$	2^{5}	32	0x20
$1 \ll 6$	2^{6}	64	0x40
$1 \ll 7$	2^{7}	128	0x80
$1 \ll 8$	2^{8}	256	0x100
$1 \ll 9$	2^{9}	512	0x200
$1 \ll 10$	2^{10}	1024	0x400
$1 \ll 11$	2^{11}	2048	0x800
$1 \ll 12$	2^{12}	4096	0x1000
$1 \ll 13$	2^{13}	8192	0x2000
$1 \ll 14$	2^{14}	16384	0x4000
$1 \ll 15$	2^{15}	32768	0x8000
$1 \ll 16$	2^{16}	65536	0x10000
$1 \ll 17$	2^{17}	131072	0x20000
$1 \ll 18$	2^{18}	262144	0x40000
$1 \ll 19$	2^{19}	524288	0x80000
$1 \ll 20$	2^{20}	1048576	0x100000
$1 \ll 21$	2^{21}	2097152	0x200000
$1 \ll 22$	2^{22}	4194304	0x400000
$1 \ll 23$	2^{23}	8388608	0x800000
$1 \ll 24$	2^{24}	16777216	0x1000000
$1 \ll 25$	2^{25}	33554432	0x2000000
$1 \ll 26$	2^{26}	67108864	0x4000000
$1 \ll 27$	2^{27}	134217728	0x8000000
$1 \ll 28$	2^{28}	268435456	0x10000000
$1 \ll 29$	2^{29}	536870912	0x20000000
$1 \ll 30$	2^{30}	1073741824	0x40000000
$1 \ll 31$	2^{31}	2147483648	0x80000000

Ces constantes (masques de bit) apparaissent très souvent le code et un rétro-ingénieur pratiquant doit pouvoir les repérer rapidement.

Les nombres décimaux avant 65536 et les hexadécimaux sont faciles à mémoriser. Tandis que les nombres décimaux après 65536 ne valent probablement pas la peine de l'être.

Ces constantes sont utilisées très souvent pour mapper des flags sur des bits spécifiques. Par exemple, voici un extrait de `ssl_private.h` du code source d'Apache 2.4.6:

```
/**
 * Define the SSL options
 */
#define SSL_OPT_NONE           (0)
#define SSL_OPT_RELSET         (1<<0)
```

```
#define SSL_OPT_STDENVVARS     (1<<1)
#define SSL_OPT_EXPORTCERTDATA (1<<3)
#define SSL_OPT_FAKEBASICAUTH  (1<<4)
#define SSL_OPT_STRICTREQUIRE  (1<<5)
#define SSL_OPT_OPTRENEGOTIATE (1<<6)
#define SSL_OPT_LEGACYDNFORMAT (1<<7)
```

Revenons à notre exemple.

La macro IS_SET teste la présence d'un bit dans a.

La macro IS_SET est en fait l'opération logique AND (*AND*) et elle renvoie 0 si le bit testé est absent (à 0), ou le masque de bit, si le bit est présent (à 1). L'opérateur *if()* en C/C++ exécute son code si l'expression n'est pas zéro, cela peut même être 123456, c'est pourquoi il fonctionne toujours correctement.

x86

MSVC

Compilons-le (MSVC 2010) :

Listing 1.291 : MSVC 2010

```
_rt$ = -8            ; taille = 4
_i$ = -4             ; taille = 4
_a$ = 8              ; taille = 4
_f  PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 8
    mov    DWORD PTR _rt$[ebp], 0
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN4@f
$LN3@f:
    mov    eax, DWORD PTR _i$[ebp]   ; incrémenter i
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN4@f:
    cmp    DWORD PTR _i$[ebp], 32    ; 00000020H
    jge    SHORT $LN2@f              ; boucle terminée?
    mov    edx, 1
    mov    ecx, DWORD PTR _i$[ebp]
    shl    edx, cl                   ; EDX=EDX<<CL
    and    edx, DWORD PTR _a$[ebp]
    je     SHORT $LN1@f              ; résultat de l'instruction AND égal à
    0?
                                     ; alors passer les instructions
    suivantes
    mov    eax, DWORD PTR _rt$[ebp]  ; non, différent de zéro
    add    eax, 1                    ; incrémenter rt
    mov    DWORD PTR _rt$[ebp], eax
$LN1@f:
    jmp    SHORT $LN3@f
```

```
$LN2@f:
    mov     eax, DWORD PTR _rt$[ebp]
    mov     esp, ebp
    pop     ebp
    ret     0
_f    ENDP
```

OllyDbg

Chargeons cet exemple dans OllyDbg. Définissons la valeur d'entrée à `0x12345678`.

Pour $i = 1$, nous voyons comment i est chargé dans ECX :

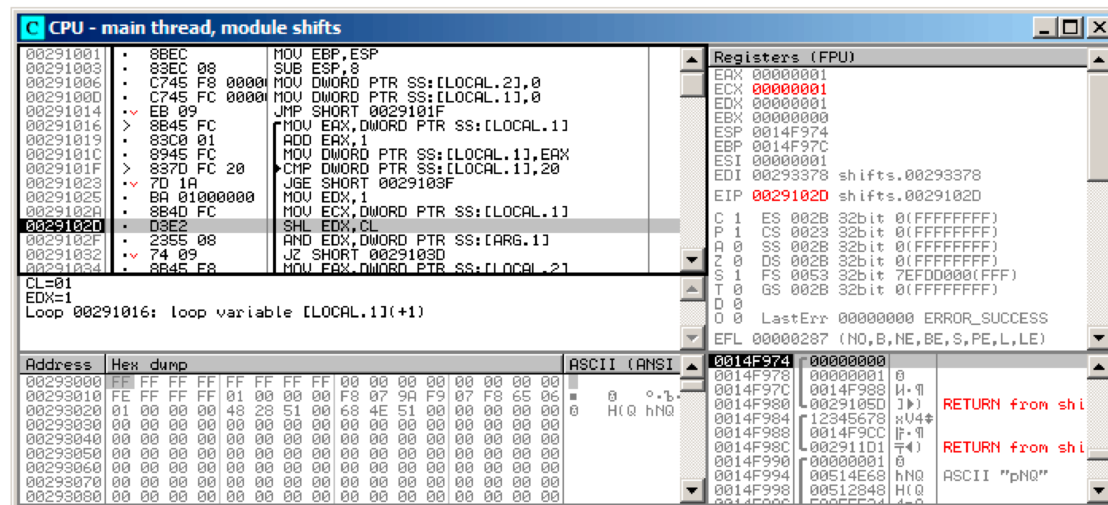


Fig. 1.98: OllyDbg : $i = 1$, i est chargé dans ECX

EDX contient 1. SHL va être exécuté maintenant.

SHL a été exécuté:

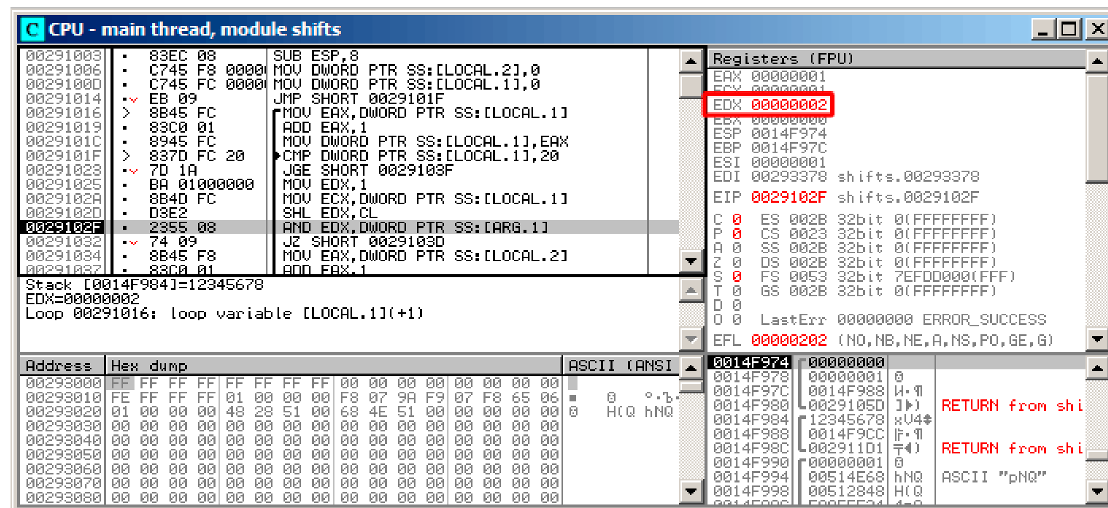


Fig. 1.99: OllyDbg : $i = 1$, EDX $=1 \ll 1 = 2$

EDX contient $1 \ll 1$ (ou 2). Ceci est un masque de bit.

AND met ZF à 1, ce qui implique que la valeur en entrée (0x12345678) ANDée avec 2 donne 0:

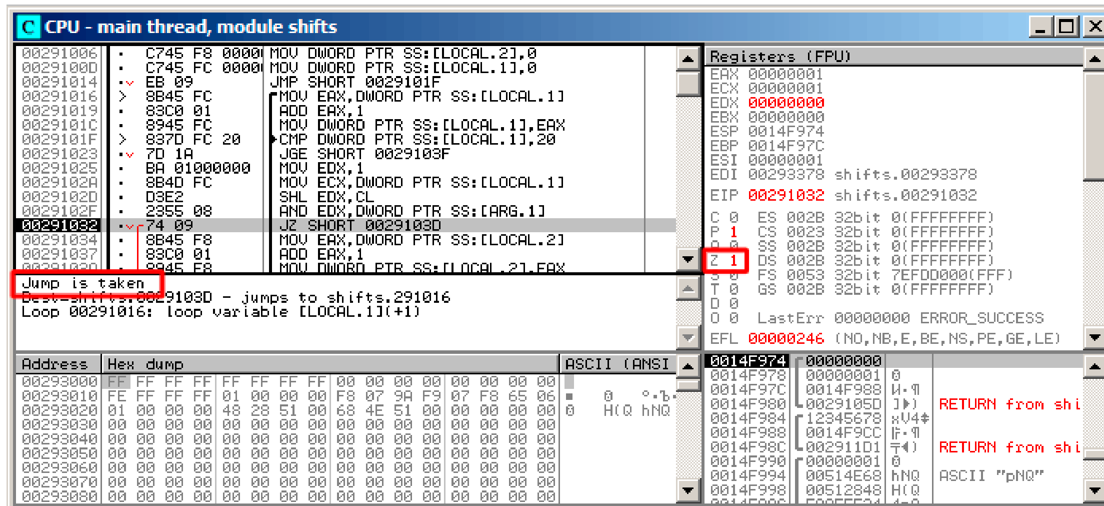


Fig. 1.100: OllyDbg : $i = 1$, y a-t-il ce bit dans la valeur en entrée? Non. (ZF =1)

Donc, il n'y a pas le bit correspondant dans la valeur en entrée.

Le morceau de code, qui incrémente le compteur ne va pas être exécuté: l'instruction JZ l'évite.

Avançons un peu plus et i vaut maintenant 4. SHL va être exécuté maintenant:

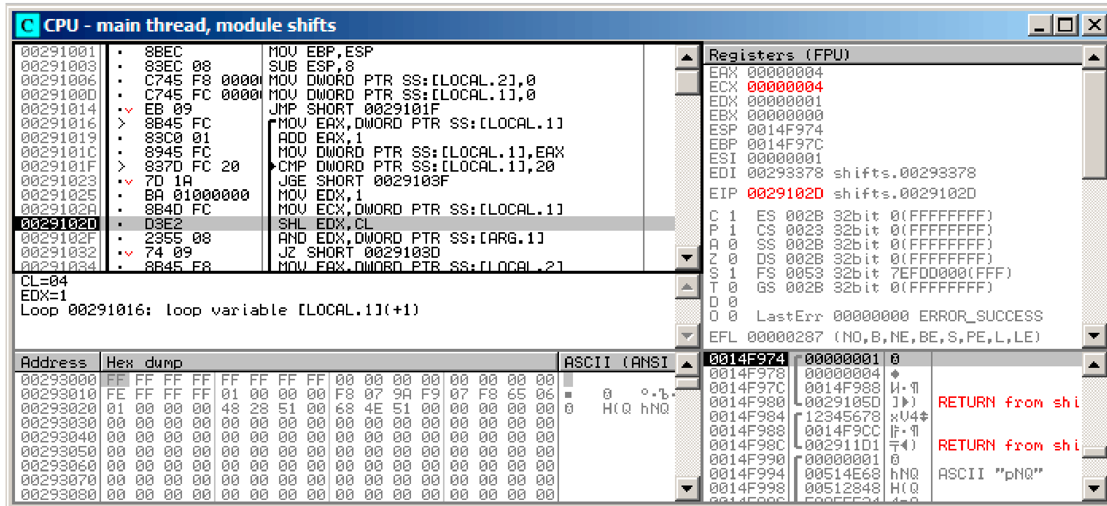


Fig. 1.101: OllyDbg : $i = 4$, i est chargée dans ECX

EDX $=1 \ll 4$ (ou 0x10 ou 16) :

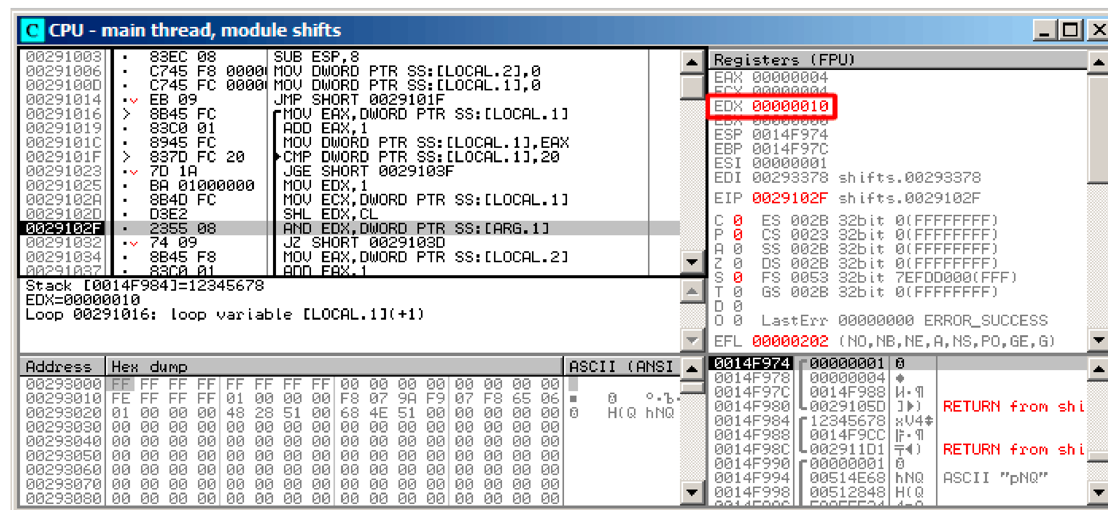


Fig. 1.102: OllyDbg : $i = 4$, EDX $=1 \ll 4 = 0x10$

Ceci est un autre masque de bit.

AND est exécuté:

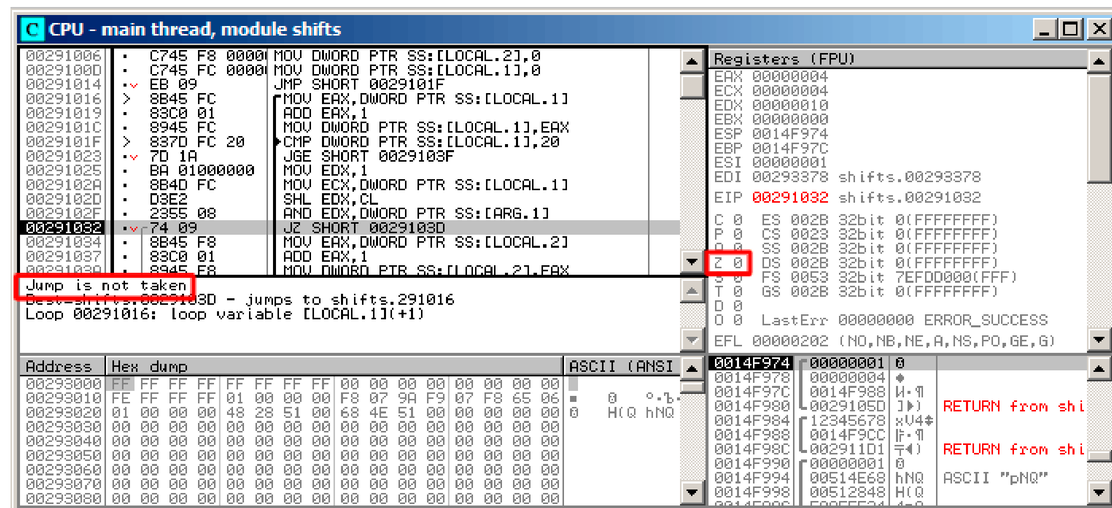


Fig. 1.103: OllyDbg : $i = 4$, y a-t-il ce bit dans la valeur en entrée? Oui. (ZF =0)

ZF est à 0 car ce bit est présent dans la valeur en entrée.
En effet, `0x12345678 & 0x10 = 0x10`.

Ce bit compte: le saut n'est pas effectué et le compteur de bit est incrémenté.

La fonction renvoie 13. C'est le nombre total de bits à 1 dans `0x12345678`.

GCC

Compilons-le avec GCC 4.4.1:

Listing 1.292 : GCC 4.4.1

```
                public f
f               proc near

rt              = dword ptr -0Ch
i               = dword ptr -8
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                push    ebx
                sub     esp, 10h
                mov     [ebp+rt], 0
                mov     [ebp+i], 0
                jmp     short loc_80483EF
loc_80483D0:
                mov     eax, [ebp+i]
```

```
                mov     edx, 1
                mov     ebx, edx
                mov     ecx, eax
                shl     ebx, cl
                mov     eax, ebx
                and     eax, [ebp+arg_0]
                test    eax, eax
                jz      short loc_80483EB
                add     [ebp+rt], 1
loc_80483EB:
                add     [ebp+i], 1
loc_80483EF:
                cmp     [ebp+i], 1Fh
                jle     short loc_80483D0
                mov     eax, [ebp+rt]
                add     esp, 10h
                pop     ebx
                pop     ebp
                retn
f               endp
```

x64

Modifions légèrement l'exemple pour l'étendre à 64-bit:

```
#include <stdio.h>
#include <stdint.h>

#define IS_SET(flag, bit)       ((flag) & (bit))

int f(uint64_t a)
{
    uint64_t i;
    int rt=0;

    for (i=0; i<64; i++)
        if (IS_SET (a, 1ULL<<i))
            rt++;

    return rt;
};
```

GCC 4.8.2 sans optimisation

Jusqu'ici, c'est facile.

Listing 1.293 : GCC 4.8.2 sans optimisation

```
f:
        push    rbp
        mov     rbp, rsp
```

```
        mov     QWORD PTR [rbp-24], rdi ; a
        mov     DWORD PTR [rbp-12], 0   ; rt=0
        mov     QWORD PTR [rbp-8], 0    ; i=0
        jmp     .L2
.L4:
        mov     rax, QWORD PTR [rbp-8]
        mov     rdx, QWORD PTR [rbp-24]
; RAX = i, RDX = a
        mov     ecx, eax
; ECX = i
        shr     rdx, cl
; RDX = RDX>>CL = a>>i
        mov     rax, rdx
; RAX = RDX = a>>i
        and     eax, 1
; EAX = EAX&1 = (a>>i)&1
        test    rax, rax
; est-ce que le dernier bit est zéro?
; passer l'instruction ADD suivante, si c'est le cas.
        je      .L3
        add     DWORD PTR [rbp-12], 1   ; rt++
.L3:
        add     QWORD PTR [rbp-8], 1    ; i++
.L2:
        cmp     QWORD PTR [rbp-8], 63   ; i<63?
        jbe     .L4                     ; sauter au début du corps de la boucle, si oui
        mov     eax, DWORD PTR [rbp-12] ; renvoyer rt
        pop     rbp
        ret
```

GCC 4.8.2 avec optimisation

Listing 1.294 : GCC 4.8.2 avec optimisation

```
f:
        xor     eax, eax        ; la variable rt sera dans le registre EAX
        xor     ecx, ecx        ; la variable i sera dans le registre ECX
.L3:
        mov     rsi, rdi        ; charger la valeur en entrée
        lea     edx, [rax+1]    ; EDX=EAX+1
; ici EDX est la nouvelle version de rt,
; qui sera écrite dans la variable rt, si le dernier bit est à 1
        shr     rsi, cl         ; RSI=RSI>>CL
        and     esi, 1          ; ESI=ESI&1
; est-ce que le dernier bit est 1? Si oui, écrire la nouvelle version de rt dans EAX
        cmovne  eax, edx
        add     rcx, 1          ; RCX++
        cmp     rcx, 64
        jne     .L3
        rep ret                 ; AKA fatret
```

Ce code est plus concis, mais a une particularité.

Dans tous les exemples que nous avons vu jusqu'ici, nous incrémentions la valeur de «rt» après la comparaison d'un bit spécifique, mais le code ici incrémente «rt» avant (ligne 6), écrivant la nouvelle valeur dans le registre EDX. Donc, si le dernier bit est à 1, l'instruction CMOVNE[148] (qui est un synonyme pour CMOVNZ[149]) *commits* la nouvelle valeur de «rt» en déplaçant EDX («valeur proposée de rt») dans EAX («rt courant» qui va être retourné à la fin).

C'est pourquoi l'incrémentation est effectuée à chaque étape de la boucle, i.e., 64 fois, sans relation avec la valeur en entrée.

L'avantage de ce code est qu'il contient seulement un saut conditionnel (à la fin de la boucle) au lieu de deux sauts (évitant l'incrément de la valeur de «rt» et à la fin de la boucle). Et cela doit s'exécuter plus vite sur les CPUs modernes avec des prédicteurs de branchement: 2.4.1 on page 589.

La dernière instruction est REP RET (opcode F3 C3) qui est aussi appelée FATRET par MSVC. C'est en quelque sorte une version optimisée de RET, qu'AMD recommande de mettre en fin de fonction, si RET se trouve juste après un saut conditionnel: [*Software Optimization Guide for AMD Family 16h Processors*, (2013)p.15] [150].

MSVC 2010 avec optimisation

Listing 1.295 : MSVC 2010 avec optimisation

```
a$ = 8
f       PROC
; RCX = valeur en entrée
        xor     eax, eax
        mov     edx, 1
        lea     r8d, QWORD PTR [rax+64]
; R8D=64
        npad    5
$LL4@f:
        test    rdx, rcx
; il n'y a pas le même bit dans la valeur en entrée?
; alors passer la prochaine instruction INC.
        je      SHORT $LN3@f
        inc     eax     ; rt++
$LN3@f:
        rol     rdx, 1  ; RDX=RDX<<1
        dec     r8      ; R8--
        jne     SHORT $LL4@f
        fatret  0
f       ENDP
```

[148]Conditional MOVe if Not Equal
[149]Conditional MOVe if Not Zero
[150]Lire aussi à ce propos: http://repzret.org/p/repzret/

Ici l'instruction ROL est utilisée au lieu de SHL, qui est en fait «rotate left / pivoter à gauche» au lieu de «shift left / décaler à gauche», mais dans cet exemple elle fonctionne tout comme SHL.

Vous pouvez en lire plus sur l'instruction de rotation ici: .1.6 on page 1342.

R8 ici est compté de 64 à 0. C'est tout comme un i inversé.

Voici une table de quelques registres pendant l'exécution:

RDX	R8
0x0000000000000001	64
0x0000000000000002	63
0x0000000000000004	62
0x0000000000000008	61
...	...
0x4000000000000000	2
0x8000000000000000	1

À la fin, nous voyons l'instruction FATRET, qui a été expliquée ici: 1.28.5 on the preceding page.

MSVC 2012 avec optimisation

Listing 1.296 : MSVC 2012 avec optimisation

```
a$ = 8
f       PROC
; RCX = valeur en entrée
        xor     eax, eax
        mov     edx, 1
        lea     r8d, QWORD PTR [rax+32]
; EDX = 1, R8D = 32
        npad    5
$LL4@f:
; pass 1 ------------------------------
        test    rdx, rcx
        je      SHORT $LN3@f
        inc     eax     ; rt++
$LN3@f:
        rol     rdx, 1  ; RDX=RDX<<1
; -------------------------------------
; pass 2 ------------------------------
        test    rdx, rcx
        je      SHORT $LN11@f
        inc     eax     ; rt++
$LN11@f:
        rol     rdx, 1  ; RDX=RDX<<1
; -------------------------------------
        dec     r8      ; R8--
        jne     SHORT $LL4@f
        fatret  0
f       ENDP
```

MSVC 2012 avec optimisation fait presque le même job que MSVC 2010 avec optimisation, mais en quelque sorte, il génère deux corps de boucles identiques et le nombre de boucles est maintenant 32 au lieu de 64.

Pour être honnête, il n'est pas possible de dire pourquoi. Une ruse d'optimisation? Peut-être est-il meilleur pour le corps de la boucle d'être légèrement plus long?

De toute façon, ce genre de code est pertinent ici pour montrer que parfois la sortie du compilateur peut être vraiment bizarre et illogique, mais fonctionner parfaitement.

ARM + avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Listing 1.297 : avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
                MOV             R1, R0
                MOV             R0, #0
                MOV             R2, #1
                MOV             R3, R0
loc_2E54
                TST             R1, R2,LSL R3 ; mettre les flags suivant R1 &
    (R2<<R3)
                ADD             R3, R3, #1    ; R3++
                ADDNE           R0, R0, #1    ; si le flag ZF est mis par 0
    TST, alors R0++
                CMP             R3, #32
                BNE             loc_2E54
                BX              LR
```

TST est la même chose que TEST en x86.

Comme noté précédemment (3.12.3 on page 639), il n'y a pas d'instruction de décalage séparée en mode ARM Toutefois, il y a ces modificateurs LSL (*Logical Shift Left / décalage logique à gauche*), LSR (*Logical Shift Right / décalage logique à droite*), ASR (*Arithmetic Shift Right décalage arithmétique à droite*), ROR (*Rotate Right / rotation à droite*) et RRX (*Rotate Right with Extend / rotation à droite avec extension*), qui peuvent être ajoutés à des instructions comme MOV, TST, CMP, ADD, SUB, RSB[151].

Ces modificateurs définissent comment décaler le second opérande et de combien de bits.

Ainsi l'instruction «TST R1, R2,LSL R3» fonctionne ici comme $R1 \wedge (R2 \ll R3)$.

ARM + avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

Presque la même, mais ici il y a deux instructions utilisées, LSL.W/TST, au lieu d'une seule TST, car en mode Thumb il n'est pas possible de définir le modificateur LSL directement dans TST.

```
                MOV             R1, R0
                MOVS            R0, #0
                MOV.W           R9, #1
```

[151] Ces instructions sont également appelées «instructions de traitement de données»

```
                MOVS            R3, #0
loc_2F7A
                LSL.W           R2, R9, R3
                TST             R2, R1
                ADD.W           R3, R3, #1
                IT NE
                ADDNE           R0, #1
                CMP             R3, #32
                BNE             loc_2F7A
                BX              LR
```

ARM64 + GCC 4.9 avec optimisation

Prenons un exemple en 64.bit qui a déjà été utilisé: 1.28.5 on page 424.

Listing 1.298 : GCC (Linaro) 4.8 avec optimisation

```
f:
        mov     w2, 0           ; rt=0
        mov     x5, 1
        mov     w1, w2
.L2:
        lsl     x4, x5, x1      ; w4 = w5<<w1 = 1<<i
        add     w3, w2, 1       ; new_rt=rt+1
        tst     x4, x0          ; (1<<i) & a
        add     w1, w1, 1       ; i++
; le résultat de TST était non-zéro?
; alors w2=w3 ou rt=new_rt.
; autrement: w2=w2 ou rt=rt (opération idle)
        csel    w2, w3, w2, ne
        cmp     w1, 64          ; i<64?
        bne     .L2             ; oui
        mov     w0, w2          ; renvoyer rt
        ret
```

Le résultat est très semblable à ce que GCC génère pour x64: 1.294 on page 425.

L'instruction CSEL signifie «Conditional SELect / sélection conditionnelle». Elle choisi une des deux variables en fonction des flags mis par TST et copie la valeur dans W2, qui contient la variable «rt».

ARM64 + GCC 4.9 sans optimisation

De nouveau, nous travaillons sur un exemple 64-bit qui a déjà été utilisé: 1.28.5 on page 424. Le code est plus verbeux, comme d'habitude.

Listing 1.299 : sans optimisation GCC (Linaro) 4.8

```
f:
        sub     sp, sp, #32
        str     x0, [sp,8]      ; stocker la valeur de "a" dans la zone de
                    ; sauvegarde des registres
        str     wzr, [sp,24]    ; rt=0
```

```
        str     wzr, [sp,28]    ; i=0
        b       .L2
.L4:
        ldr     w0, [sp,28]
        mov     x1, 1
        lsl     x0, x1, x0      ; X0 = X1<<X0 = 1<<i
        mov     x1, x0
; X1 = 1<<i
        ldr     x0, [sp,8]
; X0 = a
        and     x0, x1, x0
; X0 = X1&X0 = (1<<i) & a
; X0 contient zéro? alors sauter en .L3, évitant d'incrémenter "rt"
        cmp     x0, xzr
        beq     .L3
; rt++
        ldr     w0, [sp,24]
        add     w0, w0, 1
        str     w0, [sp,24]
.L3:
; i++
        ldr     w0, [sp,28]
        add     w0, w0, 1
        str     w0, [sp,28]
.L2:
; i<=63? alors sauter en .L4
        ldr     w0, [sp,28]
        cmp     w0, 63
        ble     .L4
; renvoyer rt
        ldr     w0, [sp,24]
        add     sp, sp, 32
        ret
```

MIPS

GCC sans optimisation

Listing 1.300 : GCC 4.4.5 sans optimisation (IDA)

```
f:
; IDA ne connaît pas le nom des variables, nous les donnons manuellement:
rt              = -0x10
i               = -0xC
var_4           = -4
a               =  0

                addiu   $sp, -0x18
                sw      $fp, 0x18+var_4($sp)
                move    $fp, $sp
                sw      $a0, 0x18+a($fp)
; initialiser les variables rt et i à zéro:
```

```
                sw      $zero, 0x18+rt($fp)
                sw      $zero, 0x18+i($fp)
; saut aux instructions de test de la boucle
                b       loc_68
                or      $at, $zero ; slot de délai de branchement, NOP

loc_20:
                li      $v1, 1
                lw      $v0, 0x18+i($fp)
                or      $at, $zero ; slot de délai de chargement, NOP
                sllv    $v0, $v1, $v0
; $v0 = 1<<i
                move    $v1, $v0
                lw      $v0, 0x18+a($fp)
                or      $at, $zero ; slot de délai de chargement, NOP
                and     $v0, $v1, $v0
; $v0 = a & (1<<i)
; est-ce que a & (1<<i) est égal à zéro? sauter en loc_58 si oui:
                beqz    $v0, loc_58
                or      $at, $zero
; il n'y pas eu de saut, cela signifie que a & (1<<i)!=0, il faut donc
    incrémenter "rt":
                lw      $v0, 0x18+rt($fp)
                or      $at, $zero ; slot de délai de chargement, NOP
                addiu   $v0, 1
                sw      $v0, 0x18+rt($fp)

loc_58:
; incrémenter i:
                lw      $v0, 0x18+i($fp)
                or      $at, $zero ; slot de délai de chargement, NOP
                addiu   $v0, 1
                sw      $v0, 0x18+i($fp)

loc_68:
; charger i et le comparer avec 0x20 (32).
; sauter en loc_20 si il vaut moins de 0x20 (32):
                lw      $v0, 0x18+i($fp)
                or      $at, $zero ; slot de délai de chargement, NOP
                slti    $v0, 0x20  # ' '
                bnez    $v0, loc_20
                or      $at, $zero ; slot de délai de branchement, NOP
; épilogue de la fonction. renvoyer rt:
                lw      $v0, 0x18+rt($fp)
                move    $sp, $fp   ; slot de délai de chargement
                lw      $fp, 0x18+var_4($sp)
                addiu   $sp, 0x18  ; slot de délai de chargement
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

C'est très verbeux: toutes les variables locales sont situées dans la pile locale et rechargées à chaque fois que l'on en a besoin.

L'instruction SLLV est «Shift Word Left Logical Variable», elle diffère de SLL seule-

ment de ce que la valeur du décalage est encodée dans l'instruction SLL (et par conséquent fixée) mais SLLV lit cette valeur depuis un registre.

GCC avec optimisation

C'est plus concis. Il y a deux instructions de décalage au lieu d'une. Pourquoi?

Il est possible de remplacer la première instruction SLLV avec une instruction de branchement inconditionnel qui saute directement au second SLLV. Mais cela ferait une autre instruction de branchement dans la fonction, et il est toujours favorable de s'en passer: 2.4.1 on page 589.

Listing 1.301 : GCC 4.4.5 avec optimisation (IDA)

```
f:
; $a0=a
; la variable rt sera dans $v0:
                move    $v0, $zero
; la variable i sera dans $v1:
                move    $v1, $zero
                li      $t0, 1
                li      $a3, 32
                sllv    $a1, $t0, $v1
; $a1 = $t0<<$v1 = 1<<i

loc_14:
                and     $a1, $a0
; $a1 = a&(1<<i)
; incrémenter i:
                addiu   $v1, 1
; sauter en loc_28 si a&(1<<i)==0 et incrémenter rt:
                beqz    $a1, loc_28
                addiu   $a2, $v0, 1
; si le saut BEQZ n'a pas été suivi, sauver la nouvelle valeur de rt dans
    $v0:
                move    $v0, $a2

loc_28:
; si i!=32, sauter en loc_14 et préparer la prochaine valeur décalée:
                bne     $v1, $a3, loc_14
                sllv    $a1, $t0, $v1
; sortir
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

1.28.6 Conclusion

Semblables aux opérateurs de décalage de C/C++ ≪ et ≫, les instructions de décalage en x86 sont SHR/SHL (pour les valeurs non-signées) et SAR/SHL (pour les valeurs signées).

Les instructions de décalages en ARM sont LSR/LSL (pour les valeurs non-signées) et ASR/LSL (pour les valeurs signées).

Il est aussi possible d'ajouter un suffixe de décalage à certaines instructions (qui sont appelées «data processing instructions/instructions de traitement de données»).

Tester un bit spécifique (connu à l'étape de compilation)

Tester si le bit 0b1000000 (0x40) est présent dans la valeur du registre:

Listing 1.302 : C/C++

```
if (input&0x40)
    ...
```

Listing 1.303 : x86

```
TEST REG, 40h
JNZ is_set
; le bit n'est pas mis (est à 0)
```

Listing 1.304 : x86

```
TEST REG, 40h
JZ is_cleared
; le bit est mis (est à 1)
```

Listing 1.305 : ARM (Mode ARM)

```
TST REG, #0x40
BNE is_set
; le bit n'est pas mis (est à 0)
```

Parfois, AND est utilisé au lieu de TEST, mais les flags qui sont mis sont les même.

Tester un bit spécifique (spécifié lors de l'exécution)

Ceci est effectué en général par ce bout de code C/C++ (décaler la valeur de n bits vers la droite, puis couper le plus petit bit) :

Listing 1.306 : C/C++

```
if ((value>>n)&1)
    ....
```

Ceci est en général implémenté en code x86 avec:

Listing 1.307 : x86

```
; REG=input_value
; CL=n
SHR REG, CL
AND REG, 1
```

Ou (décaler 1 bit n fois à gauche, isoler ce bit dans la valeur entrée et tester si ce n'est pas zéro) :

Listing 1.308 : C/C++

```
if (value & (1<<n))
    ....
```

Ceci est en général implémenté en code x86 avec:

Listing 1.309 : x86

```
; CL=n
MOV REG, 1
SHL REG, CL
AND input_value, REG
```

Mettre à 1 un bit spécifique (connu à l'étape de compilation)

Listing 1.310 : C/C++

```
value=value|0x40;
```

Listing 1.311 : x86

```
OR REG, 40h
```

Listing 1.312 : ARM (Mode ARM) and ARM64

```
ORR R0, R0, #0x40
```

Mettre à 1 un bit spécifique (spécifié lors de l'exécution)

Listing 1.313 : C/C++

```
value=value|(1<<n);
```

Ceci est en général implémenté en code x86 avec:

Listing 1.314 : x86

```
; CL=n
MOV REG, 1
SHL REG, CL
OR input_value, REG
```

Mettre à 0 un bit spécifique (connu à l'étape de compilation)

Il suffit d'effectuer l'opération AND sur la valeur inversée:

Listing 1.315 : C/C++

```
value=value&(~0x40);
```

Listing 1.316 : x86

```
AND REG, 0FFFFFFBFh
```

Listing 1.317 : x64

```
AND REG, 0FFFFFFFFFFFFFFBFh
```

Ceci laisse tous les bits qui sont à 1 inchangés excepté un.

ARM en mode ARM a l'instruction BIC, qui fonctionne comme la paire d'instructions: NOT +AND :

Listing 1.318 : ARM (Mode ARM)

```
BIC R0, R0, #0x40
```

Mettre à 0 un bit spécifique (spécifié lors de l'exécution)

Listing 1.319 : C/C++

```
value=value&(~(1<<n));
```

Listing 1.320 : x86

```
; CL=n
MOV REG, 1
SHL REG, CL
NOT REG
AND input_value, REG
```

1.28.7 Exercices

- http://challenges.re/67
- http://challenges.re/68
- http://challenges.re/69
- http://challenges.re/70

1.29 Générateur congruentiel linéaire comme générateur de nombres pseudo-aléatoires

Peut-être que le générateur congruentiel linéaire est le moyen le plus simple possible de générer des nombres aléatoires.

Ce n'est plus très utilisé aujourd'hui[152], mais il est si simple (juste une multiplication, une addition et une opération AND) que nous pouvons l'utiliser comme un exemple.

```
#include <stdint.h>

// constantes du livre Numerical Recipes
#define RNG_a 1664525
#define RNG_c 1013904223

static uint32_t rand_state;

void my_srand (uint32_t init)
{
        rand_state=init;
}

int my_rand ()
{
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
}
```

Il y a deux fonctions: la première est utilisée pour initialiser l'état interne, et la seconde est appelée pour générer un nombre pseudo-aléatoire.

Nous voyons que deux constantes sont utilisées dans l'algorithme. Elles proviennent de [William H. Press and Saul A. Teukolsky and William T. Vetterling and Brian P. Flannery, *Numerical Recipes*, (2007)].

Définissons-les en utilisant la déclaration C/C++ `#define`. C'est une macro.

La différence entre une macro C/C++ et une constante est que toutes les macros sont remplacées par leur valeur par le pré-processeur C/C++, et qu'elles n'utilisent pas de mémoire, contrairement aux variables.

Par contre, une constante est une variable en lecture seule.

Il est possible de prendre un pointeur (ou une adresse) d'une variable constante, mais c'est impossible de faire ça avec une macro.

La dernière opération AND est nécessaire car d'après le standard C `my_rand()` doit renvoyer une valeur dans l'intervalle 0..32767.

Si vous voulez obtenir des valeurs pseudo-aléatoires 32-bit, il suffit d'omettre la dernière opération AND.

[152] Le twister de Mersenne est meilleur.

1.29.1 x86

Listing 1.321 : MSVC 2013 avec optimisation

```
_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
        mov     eax, DWORD PTR _init$[esp-4]
        mov     DWORD PTR _rand_state, eax
        ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
        imul    eax, DWORD PTR _rand_state, 1664525
        add     eax, 1013904223 ; 3c6ef35fH
        mov     DWORD PTR _rand_state, eax
        and     eax, 32767      ; 00007fffH
        ret     0
_rand   ENDP

_TEXT   ENDS
```

Nous les voyons ici: les deux constantes sont intégrées dans le code. Il n'y a pas de mémoire allouée pour elles.

La fonction `my_srand()` copie juste sa valeur en entrée dans la variable `rand_state` interne.

`my_rand()` la prend, calcule le `rand_state` suivant, le coupe et le laisse dans le registre EAX.

La version non optimisée est plus verbeuse:

Listing 1.322 : MSVC 2013 sans optimisation

```
_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _init$[ebp]
        mov     DWORD PTR _rand_state, eax
        pop     ebp
        ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
```

```
        push    ebp
        mov     ebp, esp
        imul    eax, DWORD PTR _rand_state, 1664525
        mov     DWORD PTR _rand_state, eax
        mov     ecx, DWORD PTR _rand_state
        add     ecx, 1013904223 ; 3c6ef35fH
        mov     DWORD PTR _rand_state, ecx
        mov     eax, DWORD PTR _rand_state
        and     eax, 32767      ; 00007fffH
        pop     ebp
        ret     0
_rand   ENDP

_TEXT   ENDS
```

1.29.2 x64

La version x64 est essentiellement la même et utilise des registres 32-bit au lieu de 64-bit (car nous travaillons avec des valeurs de type *int* ici).

Mais `my_srand()` prend son argument en entrée dans le registre ECX plutôt que sur la pile:

Listing 1.323 : MSVC 2013 x64 avec optimisation

```
_BSS    SEGMENT
rand_state DD   01H DUP (?)
_BSS    ENDS

init$ = 8
my_srand PROC
; ECX = argument en entrée
        mov     DWORD PTR rand_state, ecx
        ret     0
my_srand ENDP

_TEXT   SEGMENT
my_rand PROC
        imul    eax, DWORD PTR rand_state, 1664525 ; 0019660dH
        add     eax, 1013904223 ; 3c6ef35fH
        mov     DWORD PTR rand_state, eax
        and     eax, 32767      ; 00007fffH
        ret     0
my_rand ENDP

_TEXT   ENDS
```

Le compilateur GCC génère en grande partie le même code.

1.29.3 ARM 32-bit

Listing 1.324 : avec optimisation Keil 6/2013 (Mode ARM)

```
my_srand PROC
        LDR      r1,|L0.52|  ; charger un pointeur sur rand_state
        STR      r0,[r1,#0]  ; sauver rand_state
        BX       lr
        ENDP

my_rand PROC
        LDR      r0,|L0.52|  ; charger un pointeur sur rand_state
        LDR      r2,|L0.56|  ; charger RNG_a
        LDR      r1,[r0,#0]  ; charger rand_state
        MUL      r1,r2,r1
        LDR      r2,|L0.60|  ; charger RNG_c
        ADD      r1,r1,r2
        STR      r1,[r0,#0]  ; sauver rand_state
; AND avec 0x7FFF:
        LSL      r0,r1,#17
        LSR      r0,r0,#17
        BX       lr
        ENDP

|L0.52|
        DCD      ||.data||
|L0.56|
        DCD      0x0019660d
|L0.60|
        DCD      0x3c6ef35f

        AREA ||.data||, DATA, ALIGN=2

rand_state
        DCD      0x00000000
```

Il n'est pas possible d'intégrer une constante 32-bit dans des instructions ARM, donc Keil doit les stocker à l'extérieur et en outre les charger. Une chose intéressante est qu'il n'est pas possible non plus d'intégrer la constante 0x7FFF. Donc ce que fait Keil est de décaler `rand_state` vers la gauche de 17 bits et ensuite la décale de 17 bits vers la droite. Ceci est analogue à la déclaration $(rand_state \ll 17) \gg 17$ en C/C++. Il semble que ça soit une opération inutile, mais ce qu'elle fait est de mettre à zéro les 17 bits hauts, laissant les 15 bits bas inchangés, et c'est notre but après tout.

Keil avec optimisation pour le mode Thumb génère essentiellement le même code.

1.29.4 MIPS

Listing 1.325 : avec optimisation GCC 4.4.5 (IDA)

```
my_srand:
; stocker $a0 dans rand_state:
                lui     $v0, (rand_state >> 16)
                jr      $ra
                sw      $a0, rand_state
```

```
my_rand:
; charger rand_state dans $v0:
                lui     $v1, (rand_state >> 16)
                lw      $v0, rand_state
                or      $at, $zero  ; slot de délai de branchement
; multiplier rand_state dans $v0 par 1664525 (RNG_a):
                sll     $a1, $v0, 2
                sll     $a0, $v0, 4
                addu    $a0, $a1, $a0
                sll     $a1, $a0, 6
                subu    $a0, $a1, $a0
                addu    $a0, $v0
                sll     $a1, $a0, 5
                addu    $a0, $a1
                sll     $a0, 3
                addu    $v0, $a0, $v0
                sll     $a0, $v0, 2
                addu    $v0, $a0
; ajouter 1013904223 (RNG_c)
; l'instruction LI est la fusion par IDA de LUI et ORI
                li      $a0, 0x3C6EF35F
                addu    $v0, $a0
; stocker dans rand_state:
                sw      $v0, (rand_state & 0xFFFF)($v1)
                jr      $ra
                andi    $v0, 0x7FFF ; slot de délai de branchement
```

Ouah, ici nous ne voyons qu'une seule constante (0x3C6EF35F ou 1013904223). Où est l'autre (1664525)?

Il semble que la multiplication soit effectuée en utilisant seulement des décalages et des additions! Vérifions cette hypothèse:

```
#define RNG_a 1664525

int f (int a)
{
        return a*RNG_a;
}
```

Listing 1.326 : GCC 4.4.5 avec optimisation (IDA)

```
f:
                sll     $v1, $a0, 2
                sll     $v0, $a0, 4
                addu    $v0, $v1, $v0
                sll     $v1, $v0, 6
                subu    $v0, $v1, $v0
                addu    $v0, $a0
                sll     $v1, $v0, 5
                addu    $v0, $v1
                sll     $v0, 3
                addu    $a0, $v0, $a0
                sll     $v0, $a0, 2
```

```
                jr      $ra
                addu    $v0, $a0, $v0 ; branch delay slot
```

En effet!

Relocations MIPS

Nous allons nous concentrer sur comment les opérations comme charger et stocker dans la mémoire fonctionnent.

Les listings ici sont produits par IDA, qui cache certains détails.

Nous allons lancer objdump deux fois: pour obtenir le listing désassemblé et aussi la liste des relogements:

Listing 1.327 : GCC 4.4.5 avec optimisation (objdump)

```
# objdump -D rand_O3.o

...

00000000 <my_srand>:
   0:   3c020000        lui     v0,0x0
   4:   03e00008        jr      ra
   8:   ac440000        sw      a0,0(v0)

0000000c <my_rand>:
   c:   3c030000        lui     v1,0x0
  10:   8c620000        lw      v0,0(v1)
  14:   00200825        move    at,at
  18:   00022880        sll     a1,v0,0x2
  1c:   00022100        sll     a0,v0,0x4
  20:   00a42021        addu    a0,a1,a0
  24:   00042980        sll     a1,a0,0x6
  28:   00a42023        subu    a0,a1,a0
  2c:   00822021        addu    a0,a0,v0
  30:   00042940        sll     a1,a0,0x5
  34:   00852021        addu    a0,a0,a1
  38:   000420c0        sll     a0,a0,0x3
  3c:   00821021        addu    v0,a0,v0
  40:   00022080        sll     a0,v0,0x2
  44:   00441021        addu    v0,v0,a0
  48:   3c043c6e        lui     a0,0x3c6e
  4c:   3484f35f        ori     a0,a0,0xf35f
  50:   00441021        addu    v0,v0,a0
  54:   ac620000        sw      v0,0(v1)
  58:   03e00008        jr      ra
  5c:   30427fff        andi    v0,v0,0x7fff

...

# objdump -r rand_O3.o

...
```

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000000 R_MIPS_HI16       .bss
00000008 R_MIPS_LO16       .bss
0000000c R_MIPS_HI16       .bss
00000010 R_MIPS_LO16       .bss
00000054 R_MIPS_LO16       .bss

...
```

Considérons les deux relogements pour la fonction `my_srand()`.

La première, pour l'adresse 0 a un type de R_MIPS_HI16 et la seconde pour l'adresse 8 a un type de R_MIPS_LO16.

Cela implique que l'adresse du début du segment .bss soit écrite dans les instructions à l'adresse 0 (partie haute de l'adresse) et 8 (partie basse de l'adresse).

La variable `rand_state` est au tout début du segment .bss.

Donc nous voyons des zéros dans les opérandes des instructions LUI et SW, car il n'y a encore rien ici— le compilateur ne sait pas quoi y écrire.

L'éditeur de liens va arranger cela, et la partie haute de l'adresse sera écrite dans l'opérande de LUI et la partie basse de l'adresse—dans l'opérande de SW.

SW va ajouter la partie basse de l'adresse avec le contenu du registre $V0 (la partie haute y est).

C'est la même histoire avec la fonction my_rand() : la relogement R_MIPS_HI16 indique à l'éditeur de liens d'écrire la partie haute.

Donc la partie haute de l'adresse de la variable rand_state se trouve dans le registre $V1.

L'instruction LW à l'adresse 0x10 ajoute les parties haute et basse et charge la valeur de la variable rand_state dans $V0.

L'instruction SW à l'adresse 0x54 fait à nouveau la somme et stocke la nouvelle valeur dans la variable globale rand_state.

IDA traite les relogements pendant le chargement, cachant ainsi ces détails, mais nous devons les garder à l'esprit.

1.29.5 Version thread-safe de l'exemple

La version thread-safe de l'exemple sera montrée plus tard: 6.2.1 on page 966.

1.30 Structures

Moyennant quelques ajustements, on peut considérer qu'une structure C/C++ n'est rien d'autre qu'un ensemble de variables, pas toutes nécessairement du même type,

et toujours stockées en mémoire côte à côte [153].

1.30.1 MSVC: exemple SYSTEMTIME

Considérons la structure win32 SYSTEMTIME[154] qui décrit un instant dans le temps. Voici comment elle est définie:

Listing 1.328 : WinBase.h

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Écrivons une fonction C pour récupérer l'instant qu'il est:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME t;
    GetSystemTime (&t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t.wYear, t.wMonth, t.wDay,
        t.wHour, t.wMinute, t.wSecond);

    return;
};
```

Le résultat de la compilation avec MSVC 2010 donne:

Listing 1.329 : MSVC 2010 /GS-

```
_t$ = -16 ; size = 16
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 16
    lea    eax, DWORD PTR _t$[ebp]
    push   eax
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  ecx, WORD PTR _t$[ebp+12] ; wSecond
    push   ecx
```

153 AKA «conteneur hétérogène»
154 MSDN: SYSTEMTIME structure

```
    movzx  edx, WORD PTR _t$[ebp+10] ; wMinute
    push   edx
    movzx  eax, WORD PTR _t$[ebp+8] ; wHour
    push   eax
    movzx  ecx, WORD PTR _t$[ebp+6] ; wDay
    push   ecx
    movzx  edx, WORD PTR _t$[ebp+2] ; wMonth
    push   edx
    movzx  eax, WORD PTR _t$[ebp] ; wYear
    push   eax
    push   OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
    call   _printf
    add    esp, 28
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

16 octets sont réservés sur la pile pour cette structure, ce qui correspond exactement à `sizeof(WORD)*8`. La structure comprend effectivement 8 variables d'un WORD chacun.

Faites attention au fait que le premier membre de la structure est le champ `wYear`. On peut donc considérer que la fonction `GetSystemTime()`[155]reçoit comme argument un pointeur sur la structure SYSTEMTIME, ou bien qu'elle reçoit un pointeur sur le champ `wYear`. Et en fait c'est exactement la même chose! `GetSystemTime()` écrit l'année courante dans à l'adresse du WORD qu'il a reçu, avance de 2 octets, écrit le mois courant et ainsi de suite.

[155] MSDN: SYSTEMTIME structure

OllyDbg

Compilons cet exemple avec MSVC 2010 et les options /GS- /MD, puis exécutons le avec OllyDbg.

Ouvrons la fenêtre des données et celle de la pile à l'adresse du premier argument fourni à la fonction GetSystemTime(), puis attendons que cette fonction se termine. Nous constatons :

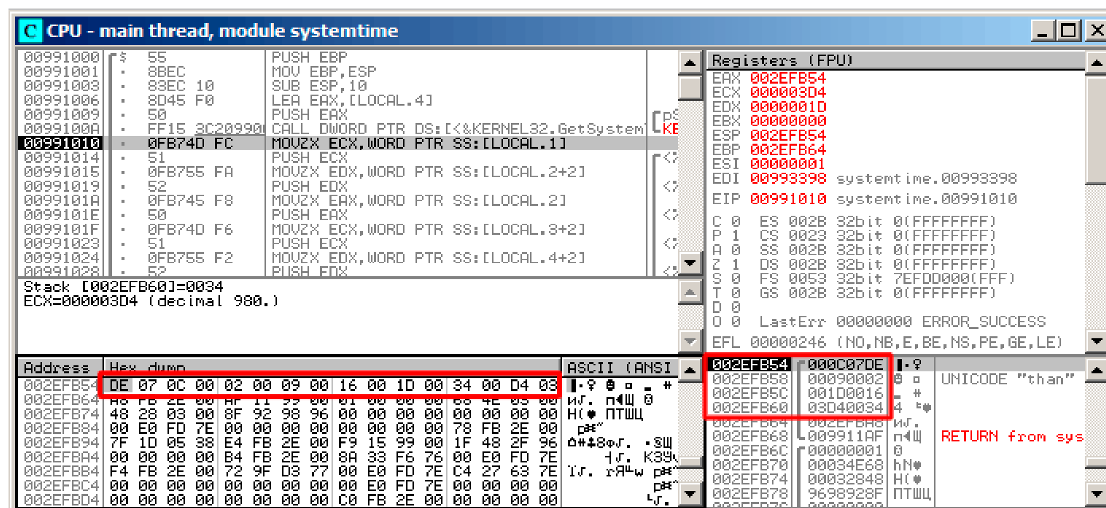


Fig. 1.104: OllyDbg : Juste après l'appel à GetSystemTime()

Sur mon ordinateur, le résultat de l'appel à la fonction est 9 décembre 2014, 22:29:52:

Listing 1.330 : printf() output

```
2014-12-09 22:29:52
```

Nous observons donc ces 16 octets dans la fenêtre de données:

```
DE 07 0C 00 02 00 09 00 16 00 1D 00 34 00 D4 03
```

Chaque paire d'octets représente l'un des champs de la structure. Puisque nous sommes en mode petit-boutien l'octet de poids faible est situé en premier, suivi de l'octet de poids fort.

Les valeurs effectivement présentes en mémoire sont donc les suivantes:

nombre hexadécimal	nombre décimal	nom du champ
0x07DE	2014	wYear
0x000C	12	wMonth
0x0002	2	wDayOfWeek
0x0009	9	wDay
0x0016	22	wHour
0x001D	29	wMinute
0x0034	52	wSecond
0x03D4	980	wMilliseconds

Les mêmes valeurs apparaissent dans la fenêtre de la pile, mais elle y sont regroupées sous forme de valeurs 32 bits.

La fonction `printf()` utilise les valeurs qui lui sont nécessaires et les affiche à la console.

Bien que certaines valeurs telles que (`wDayOfWeek` et `wMilliseconds`) ne soient pas affichées par `printf()`, elles sont bien présentes en mémoire, prêtes à être utilisées.

Remplacer la structure par un tableau

Le fait que les champs d'une structure ne sont que des variables situées côte-à-côte peut être aisément démontré de la manière suivante. Tout en conservant à l'esprit la description de la structure SYSTEMTIME, il est possible de réécrire cet exemple simple de la manière suivante:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD array[8];
    GetSystemTime (array);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        array[0] /* wYear */, array[1] /* wMonth */, array[3] /* wDay */,
        array[4] /* wHour */, array[5] /* wMinute */, array[6] /* wSecond ⤦
    ↳ */);

    return;
};
```

Le compilateur ronchonne certes un peu:

```
systemtime2.c(7) : warning C4133: 'function' : incompatible types - from '⤦
    ↳ WORD [8]' to 'LPSYSTEMTIME'
```

Mais, il consent quand même à produire le code suivant:

Listing 1.331 : sans optimisation MSVC 2010

```
$SG78573 DB     '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
```

```
_array$ = -16   ; size = 16
_main   PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 16
        lea     eax, DWORD PTR _array$[ebp]
        push    eax
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   ecx, WORD PTR _array$[ebp+12] ; wSecond
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+10] ; wMinute
        push    edx
        movzx   eax, WORD PTR _array$[ebp+8] ; wHoure
        push    eax
        movzx   ecx, WORD PTR _array$[ebp+6] ; wDay
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+2] ; wMonth
        push    edx
        movzx   eax, WORD PTR _array$[ebp] ; wYear
        push    eax
        push    OFFSET $SG78573 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
        call    _printf
        add     esp, 28
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main   ENDP
```

Qui fonctionne à l'identique du précédent!

Il est extrêmement intéressant de constater que le code assembleur produit est impossible à distinguer de celui produit par la compilation précédente.

Et ainsi celui qui observe ce code assembleur est incapable de décider avec certitude si une structure ou un tableau était déclaré dans le code source en C.

Cela étant, aucun esprit sain ne s'amuserait à déclarer un tableau ici. Car il faut aussi compter avec la possibilité que la structure soit modifiée par les développeurs, que les champs soient triés dans un autre ordre ...

Nous n'étudierons pas cet exemple avec OllyDbg, car les résultats seraient identiques à ceux que nous avons observé en utilisant la structure.

1.30.2 Allouons de l'espace pour une structure avec malloc()

Il est parfois plus simple de placer une structure sur le tas que sur la pile:

```
#include <windows.h>
#include <stdio.h>

void main()
{
```

```
    SYSTEMTIME *t;

    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t->wYear, t->wMonth, t->wDay,
        t->wHour, t->wMinute, t->wSecond);

    free (t);

    return;
};
```

Compilons cet exemple en utilisant l'option (/0x) qui facilitera nos observations.

Listing 1.332 : MSVC avec optimisation

```
_main       PROC
    push   esi
    push   16
    call   _malloc
    add    esp, 4
    mov    esi, eax
    push   esi
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  eax, WORD PTR [esi+12] ; wSecond
    movzx  ecx, WORD PTR [esi+10] ; wMinute
    movzx  edx, WORD PTR [esi+8] ; wHour
    push   eax
    movzx  eax, WORD PTR [esi+6] ; wDay
    push   ecx
    movzx  ecx, WORD PTR [esi+2] ; wMonth
    push   edx
    movzx  edx, WORD PTR [esi] ; wYear
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG78833
    call   _printf
    push   esi
    call   _free
    add    esp, 32
    xor    eax, eax
    pop    esi
    ret    0
_main       ENDP
```

Puisque `sizeof(SYSTEMTIME) = 16` c'est aussi le nombre d'octets qui doit être alloué par `malloc()`. Celui-ci renvoie dans le registre EAX un pointeur vers un bloc mémoire fraîchement alloué. Puis le pointeur est copié dans le registre ESI. La fonction win32 `GetSystemTime()` prend soin que la valeur de ESI soit la même à l'issue

de la fonction que lors de son appel. C'est pourquoi nous pouvons continuer à l'utiliser après sans avoir eu besoin de le sauvegarder.

Tiens, une nouvelle instruction —MOVZX (*Move with Zero eXtend*). La plupart du temps, elle peut être utilisée comme MOVSX. La différence est qu'elle positionne systématiquement les bits supplémentaires à 0. Elle est utilisée ici car printf() attend une valeur sur 32 bits et que nous ne disposons que d'un WORD dans la structure — c'est à dire une valeur non signée sur 16 bits. Il nous faut donc forcer à zéro les bits 16 à 31 lorsque le WORD est copié dans un *int*, sinon nous risquons de récupérer des bits résiduels de la précédente opération sur le registre.

Dans cet exemple, il reste possible de représenter la structure sous forme d'un tableau de 8 WORDs:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD *t;

    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
        t[4] /* wHour */, t[5] /* wMinute */, t[6] /* wSecond */);

    free (t);

    return;
};
```

Nous avons alors:

Listing 1.333 : MSVC avec optimisation

```
$SG78594 DB     '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_main   PROC
        push    esi
        push    16
        call    _malloc
        add     esp, 4
        mov     esi, eax
        push    esi
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   eax, WORD PTR [esi+12]
        movzx   ecx, WORD PTR [esi+10]
        movzx   edx, WORD PTR [esi+8]
        push    eax
        movzx   eax, WORD PTR [esi+6]
        push    ecx
```

```
        movzx   ecx, WORD PTR [esi+2]
        push    edx
        movzx   edx, WORD PTR [esi]
        push    eax
        push    ecx
        push    edx
        push    OFFSET $SG78594
        call    _printf
        push    esi
        call    _free
        add     esp, 32
        xor     eax, eax
        pop     esi
        ret     0
_main   ENDP
```

Encore une fois nous obtenons un code qu'il n'est pas possible de discerner du précédent.

Et encore une fois, vous n'avez pas intérêt à faire cela, sauf si vous savez exactement ce que vous faites.

1.30.3 UNIX: struct tm

Linux

Prenons pour exemple la structure `tm` dans l'en-tête `time.h` de Linux:

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    printf ("Year: %d\n", t.tm_year+1900);
    printf ("Month: %d\n", t.tm_mon);
    printf ("Day: %d\n", t.tm_mday);
    printf ("Hour: %d\n", t.tm_hour);
    printf ("Minutes: %d\n", t.tm_min);
    printf ("Seconds: %d\n", t.tm_sec);
};
```

Compilons l'exemple avec GCC 4.4.1:

Listing 1.334 : GCC 4.4.1

```
main proc near
```

```
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h
    sub     esp, 40h
    mov     dword ptr [esp], 0 ; premier argument de la fonction time()
    call    time
    mov     [esp+3Ch], eax
    lea     eax, [esp+3Ch]  ; récupération de la valeur retournée par time()
    lea     edx, [esp+10h]  ; la structure tm est à l'adresse ESP+10h
    mov     [esp+4], edx    ; passons le pointeur vers la structure begin
    mov     [esp], eax      ; ... et le pointeur retourné par time()
    call    localtime_r
    mov     eax, [esp+24h]  ; tm_year
    lea     edx, [eax+76Ch] ; edx=eax+1900
    mov     eax, offset format ; "Year: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+20h]      ; tm_mon
    mov     eax, offset aMonthD ; "Month: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+1Ch]     ; tm_mday
    mov     eax, offset aDayD  ; "Day: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+18h]     ; tm_hour
    mov     eax, offset aHourD ; "Hour: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+14h]        ; tm_min
    mov     eax, offset aMinutesD ; "Minutes: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+10h]
    mov     eax, offset aSecondsD ; "Seconds: %d\n"
    mov     [esp+4], edx          ; tm_sec
    mov     [esp], eax
    call    printf
    leave
    retn
main endp
```

IDA n'a pas utilisé le nom des variables locales pour identifier les éléments de la pile. Mais comme nous sommes déjà des rétro ingénieurs expérimentés :-) nous pouvons nous en passer dans cet exemple simple.

Notez l'instruction `lea edx, [eax+76Ch]` —qui incrémente la valeur de EAX de `0x76C` (1900) sans modifier aucun des drapeaux. Référez-vous également à la section au

sujet de LEA (.1.6 on page 1333).

GDB

Tentons de charger l'exemple dans GDB [156] :

Listing 1.335 : GDB

```
dennis@ubuntuvm:~/polygon$ date
Mon Jun  2 18:10:37 EEST 2014
dennis@ubuntuvm:~/polygon$ gcc GCC_tm.c -o GCC_tm
dennis@ubuntuvm:~/polygon$ gdb GCC_tm
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/GCC_tm...(no debugging symbols ↙
    ↳ found)...done.
(gdb) b printf
Breakpoint 1 at 0x8048330
(gdb) run
Starting program: /home/dennis/polygon/GCC_tm

Breakpoint 1, __printf (format=0x80485c0 "Year: %d\n") at printf.c:29
29      printf.c: No such file or directory.
(gdb) x/20x $esp
0xbffff0dc:     0x080484c3      0x080485c0      0x000007de      0x00000000
0xbffff0ec:     0x08048301      0x538c93ed      0x00000025      0x0000000a
0xbffff0fc:     0x00000012      0x00000002      0x00000005      0x00000072
0xbffff10c:     0x00000001      0x00000098      0x00000001      0x00002a30
0xbffff11c:     0x0804b090      0x08048530      0x00000000      0x00000000
(gdb)
```

Nous retrouvons facilement notre structure dans la pile. Commençons par observer sa définition dans *time.h* :

Listing 1.336 : time.h

```
struct tm
{
  int   tm_sec;
  int   tm_min;
  int   tm_hour;
  int   tm_mday;
  int   tm_mon;
  int   tm_year;
  int   tm_wday;
  int   tm_yday;
  int   tm_isdst;
};
```

Faites attention au fait qu'ici les champs sont des *int* sur 32 bits et non des WORD comme dans SYSTEMTIME.

[156]Le résultat *date* est légèrement modifié pour les besoins de la démonstration, car il est bien entendu impossible d'exécuter GDB aussi rapidement.

Voici donc les champs de notre structure tels qu'ils sont présents dans la pile:

```
0xbffff0dc: 0x080484c3      0x080485c0      0x000007de       0x00000000
0xbffff0ec: 0x08048301      0x538c93ed      0x00000025 sec   0x0000000a min
0xbffff0fc: 0x00000012 hour 0x00000002 mday 0x00000005 mon   0x00000072 ↙
    ↳ year
0xbffff10c: 0x00000001 wday 0x00000098 yday 0x00000001 isdst 0x00002a30
0xbffff11c: 0x0804b090      0x08048530      0x00000000       0x00000000
```

Représentés sous forme tabulaire, cela donne:

Hexadécimal	Décimal	nom
0x00000025	37	tm_sec
0x0000000a	10	tm_min
0x00000012	18	tm_hour
0x00000002	2	tm_mday
0x00000005	5	tm_mon
0x00000072	114	tm_year
0x00000001	1	tm_wday
0x00000098	152	tm_yday
0x00000001	1	tm_isdst

C'est très similaire à SYSTEMTIME (1.30.1 on page 443), Là encore certains champs sont présents qui ne sont pas utilisés tels que tm_wday, tm_yday, tm_isdst.

ARM

avec optimisation Keil 6/2013 (Mode Thumb)

Même exemple:

Listing 1.337 : avec optimisation Keil 6/2013 (Mode Thumb)

```
var_38 = -0x38
var_34 = -0x34
var_30 = -0x30
var_2C = -0x2C
var_28 = -0x28
var_24 = -0x24
timer  = -0xC

       PUSH    {LR}
       MOVS    R0, #0          ; timer
       SUB     SP, SP, #0x34
       BL      time
       STR     R0, [SP,#0x38+timer]
       MOV     R1, SP          ; tp
       ADD     R0, SP, #0x38+timer ; timer
       BL      localtime_r
       LDR     R1, =0x76C
       LDR     R0, [SP,#0x38+var_24]
       ADDS    R1, R0, R1
```

```
        ADR     R0, aYearD      ; "Year: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_28]
        ADR     R0, aMonthD     ; "Month: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_2C]
        ADR     R0, aDayD       ; "Day: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_30]
        ADR     R0, aHourD      ; "Hour: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_34]
        ADR     R0, aMinutesD   ; "Minutes: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_38]
        ADR     R0, aSecondsD   ; "Seconds: %d\n"
        BL      __2printf
        ADD     SP, SP, #0x34
        POP     {PC}
```

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

IDA reconnaît la structure `tm` (car le logiciel a connaissance des arguments attendus par les fonctions de la librairie telles que `localtime_r()`),

Il peut donc afficher les éléments de la structure ainsi que leurs noms.

Listing 1.338 : avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
var_38 = -0x38
var_34 = -0x34

        PUSH {R7,LR}
        MOV  R7, SP
        SUB  SP, SP, #0x30
        MOVS R0, #0  ; time_t *
        BLX  _time
        ADD  R1, SP, #0x38+var_34 ; struct tm *
        STR  R0, [SP,#0x38+var_38]
        MOV  R0, SP  ; time_t *
        BLX  _localtime_r
        LDR  R1, [SP,#0x38+var_34.tm_year]
        MOV  R0, 0xF44 ; "Year: %d\n"
        ADD  R0, PC  ; char *
        ADDW R1, R1, #0x76C
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_mon]
        MOV  R0, 0xF3A ; "Month: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_mday]
        MOV  R0, 0xF35 ; "Day: %d\n"
```

```
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_hour]
        MOV  R0, 0xF2E ; "Hour: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_min]
        MOV  R0, 0xF28 ; "Minutes: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34]
        MOV  R0, 0xF25 ; "Seconds: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        ADD  SP, SP, #0x30
        POP  {R7,PC}

...

00000000 tm        struc ; (sizeof=0x2C, standard type)
00000000 tm_sec    DCD ?
00000004 tm_min    DCD ?
00000008 tm_hour   DCD ?
0000000C tm_mday   DCD ?
00000010 tm_mon    DCD ?
00000014 tm_year   DCD ?
00000018 tm_wday   DCD ?
0000001C tm_yday   DCD ?
00000020 tm_isdst  DCD ?
00000024 tm_gmtoff DCD ?
00000028 tm_zone   DCD ? ; offset
0000002C tm        ends
```

MIPS

Listing 1.339 : avec optimisation GCC 4.4.5 (IDA)

```
main:

; Le nommage des champs dans les structures a été effectué manuellement car
    IDA ne les connaît pas:

var_40          = -0x40
var_38          = -0x38
seconds         = -0x34
minutes         = -0x30
hour            = -0x2C
day             = -0x28
month           = -0x24
year            = -0x20
var_4           = -4

  lui     $gp, (__gnu_local_gp >> 16)
```

```
addiu   $sp, -0x50
la      $gp, (__gnu_local_gp & 0xFFFF)
sw      $ra, 0x50+var_4($sp)
sw      $gp, 0x50+var_40($sp)
lw      $t9, (time & 0xFFFF)($gp)
or      $at, $zero ; Gaspillage par NOP du délai de branchement
jalr    $t9
move    $a0, $zero ; Gaspillage par NOP du délai de branchement
lw      $gp, 0x50+var_40($sp)
addiu   $a0, $sp, 0x50+var_38
lw      $t9, (localtime_r & 0xFFFF)($gp)
addiu   $a1, $sp, 0x50+seconds
jalr    $t9
sw      $v0, 0x50+var_38($sp) ; Utilisation du délai de branchement
lw      $gp, 0x50+var_40($sp)
lw      $a1, 0x50+year($sp)
lw      $t9, (printf & 0xFFFF)($gp)
la      $a0, $LC0        # "Year: %d\n"
jalr    $t9
addiu   $a1, 1900 ; branch delay slot
lw      $gp, 0x50+var_40($sp)
lw      $a1, 0x50+month($sp)
lw      $t9, (printf & 0xFFFF)($gp)
lui     $a0, ($LC1 >> 16)  # "Month: %d\n"
jalr    $t9
la      $a0, ($LC1 & 0xFFFF)  # "Month: %d\n" ; Utilisation du délai de branchement
lw      $gp, 0x50+var_40($sp)
lw      $a1, 0x50+day($sp)
lw      $t9, (printf & 0xFFFF)($gp)
lui     $a0, ($LC2 >> 16)  # "Day: %d\n"
jalr    $t9
la      $a0, ($LC2 & 0xFFFF)  # "Day: %d\n" ; Utilisation du délai de branchement
lw      $gp, 0x50+var_40($sp)
lw      $a1, 0x50+hour($sp)
lw      $t9, (printf & 0xFFFF)($gp)
lui     $a0, ($LC3 >> 16)  # "Hour: %d\n"
jalr    $t9
la      $a0, ($LC3 & 0xFFFF)  # "Hour: %d\n" ; Utilisation du délai de branchement
lw      $gp, 0x50+var_40($sp)
lw      $a1, 0x50+minutes($sp)
lw      $t9, (printf & 0xFFFF)($gp)
lui     $a0, ($LC4 >> 16)  # "Minutes: %d\n"
jalr    $t9
la      $a0, ($LC4 & 0xFFFF)  # "Minutes: %d\n" ; Utilisation du délai de branchement
lw      $gp, 0x50+var_40($sp)
lw      $a1, 0x50+seconds($sp)
lw      $t9, (printf & 0xFFFF)($gp)
lui     $a0, ($LC5 >> 16)  # "Seconds: %d\n"
jalr    $t9
la      $a0, ($LC5 & 0xFFFF)  # "Seconds: %d\n" ; Utilisation du délai de branchement
```

```
66   lw      $ra, 0x50+var_4($sp)
67   or      $at, $zero ; Gaspillage par NOP du délai de branchement
68   jr      $ra
69   addiu   $sp, 0x50
70 
71 $LC0:           .ascii "Year: %d\n"<0>
72 $LC1:           .ascii "Month: %d\n"<0>
73 $LC2:           .ascii "Day: %d\n"<0>
74 $LC3:           .ascii "Hour: %d\n"<0>
75 $LC4:           .ascii "Minutes: %d\n"<0>
76 $LC5:           .ascii "Seconds: %d\n"<0>
```

Dans cet exemple, le retard à l'exécution des instructions de branchement peuvent nous égarer.

L'instruction `addiu $a1, 1900` en ligne 35 qui ajoute la valeur 1900 à l'année en est un exemple. N'oubliez pas qu'elle est exécutée avant que le l'instruction JALR ne fasse son effet.

Structure comme un ensemble de valeurs

Afin d'illustrer le fait qu'une structure n'est qu'une collection de variables située côte-à-côte, retravaillons notre exemple sur la base de la définition de la structure *tm* : listado.1.336.

```
#include <stdio.h>
#include <time.h>

void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday, tm_yday
    ↪ , tm_isdst;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &tm_sec);

    printf ("Year: %d\n", tm_year+1900);
    printf ("Month: %d\n", tm_mon);
    printf ("Day: %d\n", tm_mday);
    printf ("Hour: %d\n", tm_hour);
    printf ("Minutes: %d\n", tm_min);
    printf ("Seconds: %d\n", tm_sec);
};
```

N.B. Le pointeur vers le champ `tm_sec` est passé comme argument de la fonction `localtime_r`, en tant que premier élément de la «structure».

Le compilateur nous alerte:

Listing 1.340 : GCC 4.7.3

```
GCC_tm2.c: In function 'main':
GCC_tm2.c:11:5: warning: passing argument 2 of 'localtime_r' from ↙
    ↳ incompatible pointer type [enabled by default]
In file included from GCC_tm2.c:2:0:
/usr/include/time.h:59:12: note: expected 'struct tm *' but argument is of ↙
    ↳ type 'int *'
```

Mais il génère cependant un fragment exécutable correspondant au code assembleur suivant:

Listing 1.341 : GCC 4.7.3

```
main      proc near

var_30    = dword ptr -30h
var_2C    = dword ptr -2Ch
unix_time = dword ptr -1Ch
tm_sec    = dword ptr -18h
tm_min    = dword ptr -14h
tm_hour   = dword ptr -10h
tm_mday   = dword ptr -0Ch
tm_mon    = dword ptr -8
tm_year   = dword ptr -4

          push    ebp
          mov     ebp, esp
          and     esp, 0FFFFFFF0h
          sub     esp, 30h
          call    __main
          mov     [esp+30h+var_30], 0 ; arg 0
          call    time
          mov     [esp+30h+unix_time], eax
          lea     eax, [esp+30h+tm_sec]
          mov     [esp+30h+var_2C], eax
          lea     eax, [esp+30h+unix_time]
          mov     [esp+30h+var_30], eax
          call    localtime_r
          mov     eax, [esp+30h+tm_year]
          add     eax, 1900
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aYearD ; "Year: %d\n"
          call    printf
          mov     eax, [esp+30h+tm_mon]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aMonthD ; "Month: %d\n"
          call    printf
          mov     eax, [esp+30h+tm_mday]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aDayD ; "Day: %d\n"
          call    printf
          mov     eax, [esp+30h+tm_hour]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aHourD ; "Hour: %d\n"
          call    printf
```

```
            mov     eax, [esp+30h+tm_min]
            mov     [esp+30h+var_2C], eax
            mov     [esp+30h+var_30], offset aMinutesD ; "Minutes: %d\n"
            call    printf
            mov     eax, [esp+30h+tm_sec]
            mov     [esp+30h+var_2C], eax
            mov     [esp+30h+var_30], offset aSecondsD ; "Seconds: %d\n"
            call    printf
            leave
            retn
main        endp
```

Ce code est similaire à ce que nous avons déjà vue et il n'est pas possible de dire si le code source original contenait une structure ou un groupe de variables.

Et cela fonctionne. Mais encore une fois ce n'est pas une bonne pratique.

En règle générale les compilateurs en l'absence d'optimisation allouent les variables sur la pile dans le même ordre que celui dans lequel elles ont été déclarées dans le code source. Pour autant, ce n'est pas une garantie.

Par ailleurs certains compilateurs peuvent vous avertir que les variables `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min` n'ont pas été initialisées avant leur utilisation, mais resteront muets au sujet de `tm_sec`

Le compilateur lui non plus ne sait pas qu'ils sont appelés à être initialisés par la fonction `localtime_r()`.

Nous avons choisi cet exemple car tous les champs de la structure sont de type *int*.

Tout ceci ne fonctionnerait pas sir les champs de la structure étaient des `WORD` de 16 bits, tel que dans le cas de la structure `SYSTEMTIME` structure—`GetSystemTime()` les initialiserait de manière erronée (puisque les variables locales sont alignées sur des frontières de 32bits). Vous en saurez plus à ce sujet dans la prochaine section: «Organisation des champs dans la structure» (1.30.4 on page 463).

Une structure n'est donc qu'un groupe de variables disposées côte-à-côte en mémoire. Nous pouvons dire que la structure est une instruction adressée au compilateur et l'obligeant à conserver le groupement des variables. Cela étant dans les toutes premières versions du langage C (avant 1972), la notion de structure n'existait pas encore [Dennis M. Ritchie, *The development of the C language*, (1993)][157].

Pas d'exemple de débogage ici. Le comportement est toujours le même.

Une structure sous forme de table de 32 bits

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
```

[157] Aussi disponible en pdf

```
    time_t unix_time;
    int i;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        int tmp=((int*)&t)[i];
        printf ("0x%08X (%d)\n", tmp, tmp);
    };
};
```

Nous n'avons qu'à utiliser l'opérateur *cast* pour transformer notre pointeur vers une structure en un tableau de *int*'s. Et cela fonctionne ! Nous avons exécuté l'exemple à 23h51m45s le 26 juillet 2014.

```
0x0000002D (45)
0x00000033 (51)
0x00000017 (23)
0x0000001A (26)
0x00000006 (6)
0x00000072 (114)
0x00000006 (6)
0x000000CE (206)
0x00000001 (1)
```

Les variables sont dans le même ordre que celui dans lequel elles apparaissent dans la définition de la structure: 1.336 on page 452.

Nous avons effectué la compilation avec:

Listing 1.342 : avec optimisation GCC 4.8.1

```
main proc near
     push    ebp
     mov     ebp, esp
     push    esi
     push    ebx
     and     esp, 0FFFFFFF0h
     sub     esp, 40h
     mov     dword ptr [esp], 0 ; timer
     lea     ebx, [esp+14h]
     call    _time
     lea     esi, [esp+38h]
     mov     [esp+4], ebx     ; tp
     mov     [esp+10h], eax
     lea     eax, [esp+10h]
     mov     [esp], eax       ; timer
     call    _localtime_r
     nop
     lea     esi, [esi+0]       ; NOP
loc_80483D8:
```

```
; EBX pointe sur la structure,
; ESI pointe sur la fin de celle-ci.
     mov     eax, [ebx] ; get 32-bit word from array
     add     ebx, 4         ; prochain champ de la structure
     mov     dword ptr [esp+4], offset a0x08xD ; "0x%08X (%d)\n"
     mov     dword ptr [esp], 1
     mov     [esp+0Ch], eax     ; passage des arguments à printf()
     mov     [esp+8], eax
     call    ___printf_chk
     cmp     ebx, esi       ; Avons-nous atteint la fin de la structure ?
     jnz     short loc_80483D8  ; non - alors passons à la prochaine valeur
     lea     esp, [ebp-8]
     pop     ebx
     pop     esi
     pop     ebp
     retn
main endp
```

En fait, l'espace dans la pile est tout d'abord traité comme une structure, puis ensuite comme un tableau.

Le pointeur sur le tableau permet même de modifier les champs de la structure.

Et encore une fois cette manière de procéder est extrêmement douteuse et pas du tout recommandée pour l'écriture d'un code qui atterrira en production.

Exercice

Tentez de modifier (en l'augmentant de 1) le numéro du mois, en traitant la structure comme s'il s'agissait d'un tableau.

Une structure sous forme d'un tableau d'octets

Nous pouvons aller plus loin. Utilisons l'opérateur *cast* pour transformer le pointeur en un tableau d'octets, puis affichons son contenu:

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i, j;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        for (j=0; j<4; j++)
```

```
            printf ("0x%02X ", ((unsigned char*)&t)[i*4+j]);
        printf ("\n");
    };
};
```

```
0x2D 0x00 0x00 0x00
0x33 0x00 0x00 0x00
0x17 0x00 0x00 0x00
0x1A 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0x72 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0xCE 0x00 0x00 0x00
0x01 0x00 0x00 0x00
```

Cet exemple a été exécuté à 23h51m45s le 26 juillet 2014 [158]. Les valeurs sont identiques à celles du précédent affichage (1.30.3 on page 460), et bien entendu l'octet de poids faible figure en premier puisque nous sommes sur une architecture de type little-endian (2.2 on page 586).

Listing 1.343 : avec optimisation GCC 4.8.1

```
main proc near
     push    ebp
     mov     ebp, esp
     push    edi
     push    esi
     push    ebx
     and     esp, 0FFFFFFF0h
     sub     esp, 40h
     mov     dword ptr [esp], 0 ; timer
     lea     esi, [esp+14h]
     call    _time
     lea     edi, [esp+38h]  ; struct end
     mov     [esp+4], esi    ; tp
     mov     [esp+10h], eax
     lea     eax, [esp+10h]
     mov     [esp], eax      ; timer
     call    _localtime_r
     lea     esi, [esi+0]       ; NOP
; ESI pointe sur la structure située sur la pile.
; EDI pointe sur la fin de la structure.
loc_8048408:
     xor     ebx, ebx        ; j=0

loc_804840A:
     movzx   eax, byte ptr [esi+ebx]     ; load byte
     add     ebx, 1          ; j=j+1
     mov     dword ptr [esp+4], offset a0x02x ; "0x%02X "
     mov     dword ptr [esp], 1
```

[158] Les dates et heures sont les mêmes dans tous les exemples. Elles ont été éditées pour la clarté de la démonstration.

```
     mov     [esp+8], eax       ; Fourniture à printf() des octets qui ont
    été chargés
     call    ___printf_chk
     cmp     ebx, 4
     jnz     short loc_804840A
; Imprime un retour chariot (CR)
     mov     dword ptr [esp], 0Ah ; c
     add     esi, 4
     call    _putchar
     cmp     esi, edi        ; Avons nous atteint la fin de la structure ?
     jnz     short loc_8048408 ; j=0
     lea     esp, [ebp-0Ch]
     pop     ebx
     pop     esi
     pop     edi
     pop     ebp
     retn
main endp
```

1.30.4 Organisation des champs dans la structure

L'arrangement des champs au sein d'une structure est un élément très important.

Prenons un exemple simple:

```
#include <stdio.h>

struct s
{
    char a;
    int b;
    char c;
    int d;
};

void f(struct s s)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
};

int main()
{
    struct s tmp;
    tmp.a=1;
    tmp.b=2;
    tmp.c=3;
    tmp.d=4;
    f(tmp);
};
```

Nous avons deux champs de type *char* (occupant chacun un octet) et deux autres — de type *int* (comportant 4 octets chacun).

x86

Le résultat de la compilation est:

Listing 1.344 : MSVC 2012 /GS- /Ob0

```
_tmp$ = -16
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 16
    mov    BYTE PTR _tmp$[ebp], 1     ; initialisation du champ a
    mov    DWORD PTR _tmp$[ebp+4], 2  ; initialisation du champ b
    mov    BYTE PTR _tmp$[ebp+8], 3   ; initialisation du champ c
    mov    DWORD PTR _tmp$[ebp+12], 4 ; initialisation du champ d
    sub    esp, 16                    ; Allocation d'espace pour la structure temporaire
    mov    eax, esp
    mov    ecx, DWORD PTR _tmp$[ebp]  ; Copie de notre structure dans la structure temporaire
    mov    DWORD PTR [eax], ecx
    mov    edx, DWORD PTR _tmp$[ebp+4]
    mov    DWORD PTR [eax+4], edx
    mov    ecx, DWORD PTR _tmp$[ebp+8]
    mov    DWORD PTR [eax+8], ecx
    mov    edx, DWORD PTR _tmp$[ebp+12]
    mov    DWORD PTR [eax+12], edx
    call    _f
    add    esp, 16
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP

_s$ = 8 ; size = 16
?f@@YAXUs@@@Z PROC ; f
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp+12]
    push   eax
    movsx  ecx, BYTE PTR _s$[ebp+8]
    push   ecx
    mov    edx, DWORD PTR _s$[ebp+4]
    push   edx
    movsx  eax, BYTE PTR _s$[ebp]
    push   eax
    push   OFFSET $SG3842
    call   _printf
    add    esp, 20
    pop    ebp
    ret    0
?f@@YAXUs@@@Z ENDP ; f
_TEXT    ENDS
```

Nous passons la structure comme un tout, mais en réalité nous pouvons constater que la structure est copiée dans un espace temporaire. De l'espace est réservé pour cela ligne 10 et les 4 champs sont copiées par les lignes de 12 … 19), puis le pointeur sur l'espace temporaire est passé à la fonction.

La structure est recopiée au cas où la fonction `f()` viendrait à en modifier le contenu. Si cela arrive, la copie de la structure qui existe dans `main()` restera inchangée.

Nous pourrions également utiliser des pointeurs C/C++. Le résulta demeurerait le même, sans qu'il soit nécessaire de procéder à la copie.

Nous observons que l'adresse de chaque champ est alignée sur un multiple de 4 octets. C'est pourquoi chaque *char* occupe 4 octets (de même qu'un *int*). Pourquoi en est-il ainsi? La réponse se situe au niveau de la CPU. Il est plus facile et performant pour elle d'accéder la mémoire et de gérer le cache de données en utilisant des adresses alignées.

En revanche ce n'est pas très économique en terme d'espace.

Tentons maintenant une compilation avec l'option (`/Zp1`) (*/Zp[n] indique qu'il faut compresser les structures en utilisant des frontières tous les n octets*).

Listing 1.345 : MSVC 2012 /GS- /Zp1

```
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 12
    mov    BYTE PTR _tmp$[ebp], 1     ; Initialisation du champ a
    mov    DWORD PTR _tmp$[ebp+1], 2  ; Initialisation du champ b
    mov    BYTE PTR _tmp$[ebp+5], 3   ; Initialisation du champ c
    mov    DWORD PTR _tmp$[ebp+6], 4  ; Initialisation du champ d
    sub    esp, 12                    ; Allocation d'espace pour la structure temporaire
    mov    eax, esp
    mov    ecx, DWORD PTR _tmp$[ebp]  ; Copie de 10 octets
    mov    DWORD PTR [eax], ecx
    mov    edx, DWORD PTR _tmp$[ebp+4]
    mov    DWORD PTR [eax+4], edx
    mov    cx, WORD PTR _tmp$[ebp+8]
    mov    WORD PTR [eax+8], cx
    call    _f
    add    esp, 12
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP

_TEXT    SEGMENT
_s$ = 8 ; size = 10
?f@@YAXUs@@@Z PROC    ; f
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp+6]
```

```
    push   eax
    movsx  ecx, BYTE PTR _s$[ebp+5]
    push   ecx
    mov    edx, DWORD PTR _s$[ebp+1]
    push   edx
    movsx  eax, BYTE PTR _s$[ebp]
    push   eax
    push   OFFSET $SG3842
    call   _printf
    add    esp, 20
    pop    ebp
    ret    0
?f@@YAXUs@@@Z ENDP    ; f
```

La structure n'occupe plus que 10 octets et chaque valeur de type *char* n'occupe plus qu'un octet. Quelles sont les conséquences ? Nous économisons de la place au prix d'un accès à ces champs moins rapide que ne pourrait le faire la CPU.

La structure est également copiée dans `main()`. Cette opération ne s'effectue pas champ par champ mais par blocs en utilisant trois instructions MOV. Et pourquoi pas 4 ?

Tout simplement parce que le compilateur a décidé qu'il était préférable d'effectuer la copie en utilisant 3 paires d'instructions MOV plutôt que de copier deux mots de 32 bits puis 2 fois un octet ce qui aurait nécessité 4 paires d'instructions MOV.

Ce type d'implémentation de la copie qui repose sur les instructions MOV plutôt que sur l'appel à la fonction `memcpy()` est très répandu. La raison en est que pour de petits blocs, cette approche est plus rapide qu'un appel à `memcpy()` : 3.14.1 on page 656.

Comme vous pouvez le deviner, si la structure est utilisée dans de nombreux fichiers sources et objets, ils doivent tous être compilés avec la même convention de compactage de la structure.

Au delà de l'option MSVC `/Zp` qui permet de définir l'alignement des champs des structures, il existe également l'option du compilateur `#pragma pack` qui peut être utilisée directement dans le code source. Elle est supportée aussi bien par MSVC[159]que pars GCC[160].

Revenons à la structure SYSTEMTIME qui contient des champs de 16 bits. Comment notre compilateur sait-il les aligner sur des frontières de 1 octet ?

Le fichier `WinNT.h` contient ces instructions:

Listing 1.346 : WinNT.h

```
#include "pshpack1.h"
```

et celles-ci:

159 MSDN: Working with Packing Structures
160 Structure-Packing Pragmas

Listing 1.347 : WinNT.h

```
#include "pshpack4.h"                    // L'alignement sur 4 octets est la
    valeur par défaut
```

Le fichier PshPack1.h ressemble à ceci:

Listing 1.348 : PshPack1.h

```
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 && !defined(_M_I86)) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#else
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

Ces instructions indiquent au compilateur comment compresser les structures définies après `#pragma pack`.

OllyDbg et les champs alignés par défaut

Examinons dans OllyDbg notre exemple lorsque les champs sont alignés par défaut sur des frontières de 4 octets:

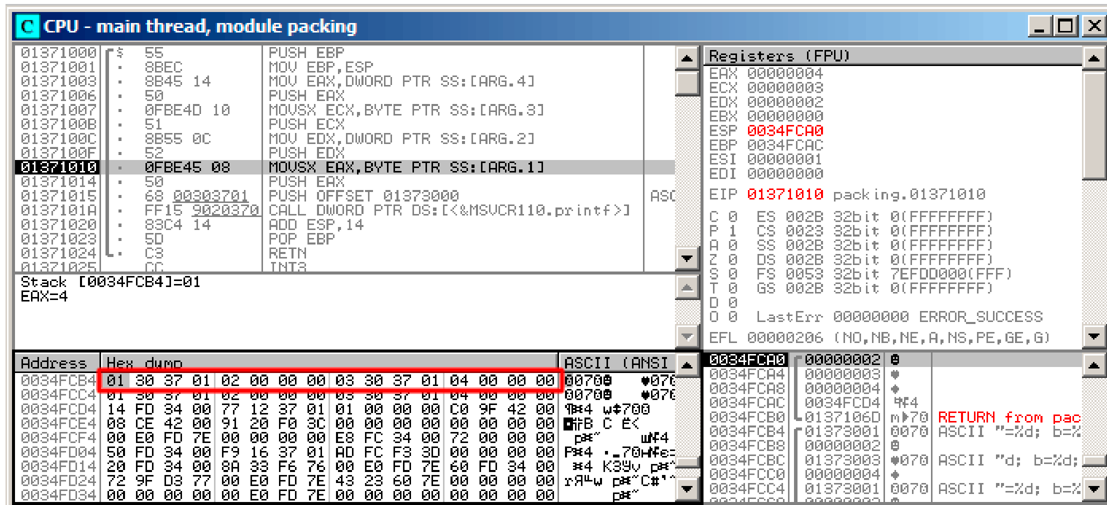


Fig. 1.105: OllyDbg : Before `printf()` execution

Nous voyons nos quatre champs dans la fenêtre de données.

Mais d'où viennent ces octets aléatoires (0x30, 0x37, 0x01) situé à côté des premier (a) et troisième (c) champs ?

Si nous revenons à notre listing 1.344 on page 464, nous constatons que ces deux champs sont de type *char*. Seul un octet est écrit pour chacun d'eux: 1 et 3 respectivement (lignes 6 et 8).

Les trois autres octets des deux mots de 32 bits ne sont pas modifiés en mémoire! Des débris aléatoires des précédentes opérations demeurent donc là.

Ces débris n'influencent nullement le résultat de la fonction `printf()` parce que les valeurs qui lui sont passées sont préparés avec l'instruction `MOVSX` qui opère sur des octets et non pas sur des mots: listado.1.344 (lignes 34 et 38).

L'instruction `MOVSX` (extension de signe) est utilisée ici car le type *char* est par défaut une valeur signée pour MSVC et GCC. Si l'un des types `unsigned char` ou `uint8_t` était utilisé ici, ce serait l'instruction `MOVZX` que le compilateur aurait choisi.

OllyDbg et les champs alignés sur des frontières de 1 octet

Les choses sont beaucoup plus simples ici. Les 4 champs occupent 10 octets et les valeurs sont stockées côte-à-côte.

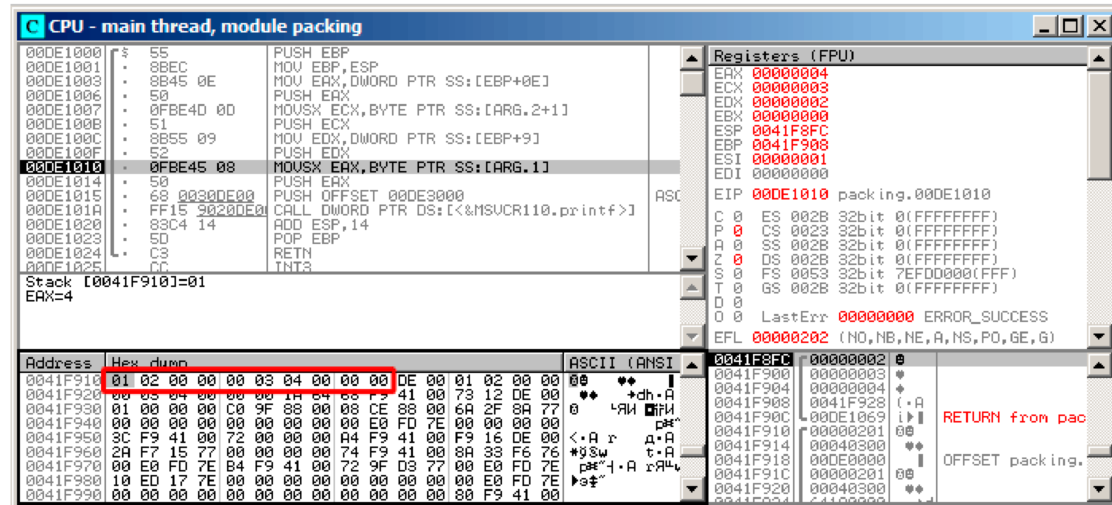


Fig. 1.106: OllyDbg : Avant appel de la fonction `printf()`

ARM

avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.349 : avec optimisation Keil 6/2013 (Mode Thumb)

```
.text:0000003E             exit ; CODE XREF: f+16
.text:0000003E 05 B0              ADD     SP, SP, #0x14
.text:00000040 00 BD              POP     {PC}


.text:00000280             f
.text:00000280
.text:00000280             var_18 = -0x18
.text:00000280             a      = -0x14
.text:00000280             b      = -0x10
.text:00000280             c      = -0xC
.text:00000280             d      = -8
.text:00000280
.text:00000280 0F B5              PUSH    {R0-R3,LR}
.text:00000282 81 B0              SUB     SP, SP, #4
.text:00000284 04 98              LDR     R0, [SP,#16]     ; d
.text:00000286 02 9A              LDR     R2, [SP,#8]      ; b
.text:00000288 00 90              STR     R0, [SP]
.text:0000028A 68 46              MOV     R0, SP
```

```
.text:0000028C 03 7B            LDRB    R3, [R0,#12]    ; c
.text:0000028E 01 79            LDRB    R1, [R0,#4]     ; a
.text:00000290 59 A0            ADR     R0, aADBDCDDD   ; "a=%d; b=%d;
    c=%d; d=%d\n"
.text:00000292 05 F0 AD FF      BL      __2printf
.text:00000296 D2 E6            B       exit
```

Rappelons-nous que c'est une structure qui est passée ici et non pas un pointeur vers une structure. Comme les 4 premiers arguments d'une fonction sont passés dans les registres sur les processeurs ARM, les champs de la structure sont passés dans les registres R0-R3.

LDRB charge un octet présent en mémoire et l'étend sur 32bits en prenant en compte son signe. Cette opération est similaire à celle effectuée par MOVSX dans les architectures x86. Elle est utilisée ici pour charger les champs a et c de la structure.

Un autre détail que nous remarquons aisément est que la fonction ne s'achève pas sur un épilogue qui lui est propre. A la place, il y a un saut vers l'épilogue d'une autre fonction! Qui plus est celui d'une fonction très différente sans aucun lien avec la nôtre. Cependant elle possède exactement le même épilogue, probablement parce qu'elle accepte utilise elle aussi 5 variables locales ($5 * 4 = 0x14$).

De plus elle est située à une adresse proche.

En réalité, peut importe l'épilogue qui est utilisé du moment que le fonctionnement est celui attendu.

Il semble donc que le compilateur Keil décide de réutiliser à des fins d'économie un fragment d'une autre fonction. Notre épilogue aurait nécessité 4 octets. L'instruction de saut n'en utilise que 2.

ARM + avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

Listing 1.350 : avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
var_C = -0xC

      PUSH   {R7,LR}
      MOV    R7, SP
      SUB    SP, SP, #4
      MOV    R9, R1 ; b
      MOV    R1, R0 ; a
      MOVW   R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d\n"
      SXTB   R1, R1  ; prepare a
      MOVT.W R0, #0
      STR    R3, [SP,#0xC+var_C] ; place d to stack for printf()
      ADD    R0, PC  ; format-string
      SXTB   R3, R2  ; prepare c
      MOV    R2, R9  ; b
      BLX    _printf
      ADD    SP, SP, #4
      POP    {R7,PC}
```

SXTB (*Signed Extend Byte*) est similaire à MOVSX pour les architectures x86. Pour le reste—c'est identique.

MIPS

Listing 1.351 : avec optimisation GCC 4.4.5 (IDA)

```
f:

var_18          = -0x18
var_10          = -0x10
var_4           = -4
arg_0           =  0
arg_4           =  4
arg_8           =  8
arg_C           =  0xC

; $a0=s.a
; $a1=s.b
; $a2=s.c
; $a3=s.d
                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x28
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x28+var_4($sp)
                sw      $gp, 0x28+var_10($sp)
; Transformation d'un octet en entier 32 bits grand-boutien (big-endian):
                sra     $t0, $a0, 24
                move    $v1, $a1
; Transformation d'un entier 32 bits grand-boutien (big-endian) en octet:
                sra     $v0, $a2, 24
                lw      $t9, (printf & 0xFFFF)($gp)
                sw      $a0, 0x28+arg_0($sp)
                lui     $a0, ($LC0 >> 16)  # "a=%d; b=%d; c=%d; d=%d\n"
                sw      $a3, 0x28+var_18($sp)
                sw      $a1, 0x28+arg_4($sp)
                sw      $a2, 0x28+arg_8($sp)
                sw      $a3, 0x28+arg_C($sp)
                la      $a0, ($LC0 & 0xFFFF)  # "a=%d; b=%d; c=%d; d=%d\n"
                move    $a1, $t0
                move    $a2, $v1
                jalr    $t9
                move    $a3, $v0 ; Gaspillage volontaire du délai de
    branchement
                lw      $ra, 0x28+var_4($sp)
                or      $at, $zero ; Gaspillage par NOP du délai de
    chargement
                jr      $ra
                addiu   $sp, 0x28 ; Gaspillage volontaire du délai de
    branchement

$LC0:           .ascii "a=%d; b=%d; c=%d; d=%d\n"<0>
```

Les champs de la structure sont fournis dans les registres $A0..$A3 puis transfor-

mé dans les registres $A1..$A3 pour l'utilisation par `printf()`, tandis que le 4ème champ (provenant de $A3) est passé sur la pile en utilisant l'instruction SW.

Mais à quoi servent ces deux instructions SRA («Shift Word Right Arithmetic») lors de la préparation des champs *char* ?

MIPS est une architecture grand-boutien (big-endian) par défaut 2.2 on page 586, de même que la distribution Debian Linux que nous utilisons.

En conséquence, lorsqu'un octet est stocké dans un emplacement 32bits d'une structure, ils occupent les bits 31..24 bits.

Quand une variable *char* doit être étendue en une valeur sur 32 bits, elle doit tout d'abord être décalée vers la droite de 24 bits.

char étant un type signé, un décalage arithmétique est utilisé ici, à la place d'un décalage logique.

Un dernier mot

Passer une structure comme argument d'une fonction (plutôt que de passer un pointeur sur cette structure) revient à passer chaque champ de la structure individuellement.

Si les champs de la structure utilisent l'alignement par défaut, la fonction f() peut être réécrite ainsi:

```
void f(char a, int b, char c, int d)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", a, b, c, d);
};
```

Le code généré par le compilateur sera le même.

1.30.5 Structures imbriquées

Maintenant qu'en est-il lorsqu'une structure est définie au sein d'une autre structure ?

```
#include <stdio.h>

struct inner_struct
{
    int a;
    int b;
};

struct outer_struct
{
    char a;
    int b;
    struct inner_struct c;
    char d;
    int e;
```

```
};

void f(struct outer_struct s)
{
    printf ("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
        s.a, s.b, s.c.a, s.c.b, s.d, s.e);
};

int main()
{
    struct outer_struct s;
    s.a=1;
    s.b=2;
    s.c.a=100;
    s.c.b=101;
    s.d=3;
    s.e=4;
    f(s);
};
```

…dans ce cas, l'ensemble des champs de `inner_struct` doivent être situés entre les champs a,b et d,e de `outer_struct`.

Compilons (MSVC 2010) :

Listing 1.352 : avec optimisation MSVC 2010 /Ob0

```
$SG2802 DB    'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0aH, 00H

_TEXT    SEGMENT
_s$ = 8
_f    PROC
    mov    eax, DWORD PTR _s$[esp+16]
    movsx  ecx, BYTE PTR _s$[esp+12]
    mov    edx, DWORD PTR _s$[esp+8]
    push   eax
    mov    eax, DWORD PTR _s$[esp+8]
    push   ecx
    mov    ecx, DWORD PTR _s$[esp+8]
    push   edx
    movsx  edx, BYTE PTR _s$[esp+8]
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG2802 ; 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d'
    call   _printf
    add    esp, 28
    ret    0
_f    ENDP

_s$ = -24
_main    PROC
    sub    esp, 24
    push   ebx
```

```
    push   esi
    push   edi
    mov    ecx, 2
    sub    esp, 24
    mov    eax, esp
; depuis ce moment, EAX est synonyme de ESP:
    mov    BYTE PTR _s$[esp+60], 1
    mov    ebx, DWORD PTR _s$[esp+60]
    mov    DWORD PTR [eax], ebx
    mov    DWORD PTR [eax+4], ecx
    lea    edx, DWORD PTR [ecx+98]
    lea    esi, DWORD PTR [ecx+99]
    lea    edi, DWORD PTR [ecx+2]
    mov    DWORD PTR [eax+8], edx
    mov    BYTE PTR _s$[esp+76], 3
    mov    ecx, DWORD PTR _s$[esp+76]
    mov    DWORD PTR [eax+12], esi
    mov    DWORD PTR [eax+16], ecx
    mov    DWORD PTR [eax+20], edi
    call   _f
    add    esp, 24
    pop    edi
    pop    esi
    xor    eax, eax
    pop    ebx
    add    esp, 24
    ret    0
_main    ENDP
```

Un point troublant est qu'en observant le code assembleur généré, nous n'avons aucun indice qui laisse penser qu'il existe une structure imbriquée! Nous pouvons donc dire que les structures imbriquées sont fusionnées avec leur conteneur pour former une seule structure *linear* ou *one-dimensional*.

Bien entendu, si nous remplaçons la déclaration `struct inner_struct c;` par `struct inner_struct *c;` (en introduisant donc un pointeur) la situation sera totalement différente.

OllyDbg

Chargeons notre exemple dans OllyDbg et observons `outer_struct` en mémoire:

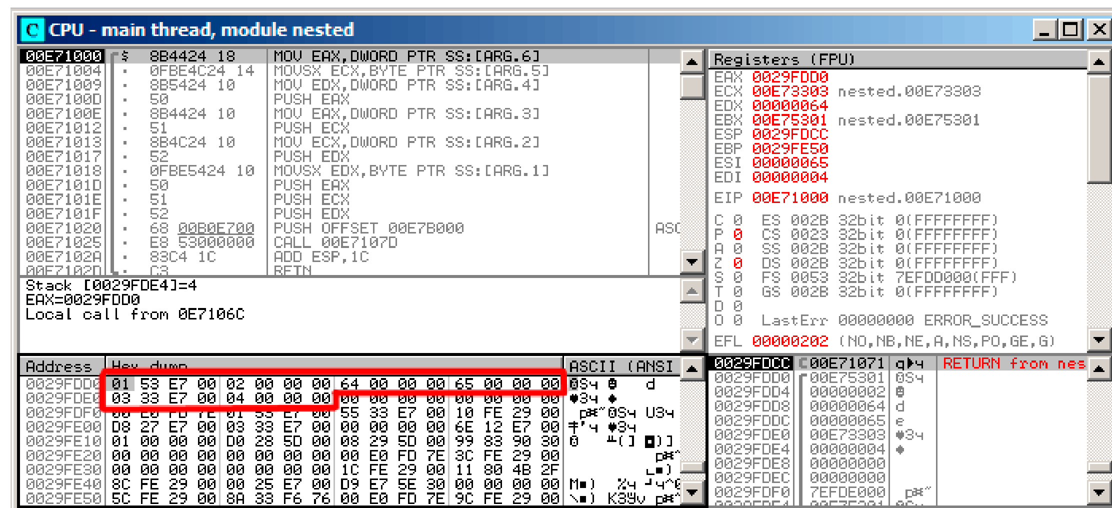


Fig. 1.107: OllyDbg : Avant appel de la fonction `printf()`

Les valeurs sont organisées en mémoire de la manière suivante:

- *(outer_struct.a)* (octet) 1 + 3 octets de détritus;
- *(outer_struct.b)* (mot de 32 bits) 2;
- *(inner_struct.a)* (mot de 32 bits) 0x64 (100);
- *(inner_struct.b)* (mot de 32 bits) 0x65 (101);
- *(outer_struct.d)* (octet) 3 + 3 octets de détritus;
- *(outer_struct.e)* (mot de 32 bits) 4.

1.30.6 Champs de bits dans une structure

Exemple CPUID

Le langage C/C++ permet de définir précisément le nombre de bits occupés par chaque champ d'une structure. Ceci est très utile lorsque l'on cherche à économise de la place. Par exemple, chaque bit permet de représenter une variable *bool*. Bien entendu, c'est au détriment de la vitesse d'exécution.

Prenons par exemple l'instruction `CPUID`[161]. Elle retourne des informations au sujet de la CPU qui exécute le programme et de ses capacités.

[161] Wikipédia

Si le registre EAX est positionné à la valeur 1 avant d'invoquer cette instruction, CPUID va retourné les informations suivantes dans le registre EAX :

3:0 (4 bits)	Stepping
7:4 (4 bits)	Modèle
11:8 (4 bits)	Famille
13:12 (2 bits)	Type de processeur
19:16 (4 bits)	Sous-modèle
27:20 (8 bits)	Sous-famille

MSVC 2010 fourni une macro CPUID, qui est absente de GCC 4.4.1. Tentons donc de rédiger nous même cette fonction pour une utilisation dans GCC grâce à l'assembleur[162]intégré à ce compilateur.

```
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
  asm volatile("cpuid":"=a"(*a),"=b"(*b),"=c"(*c),"=d"(*d):"a"(code));
}
#endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX
{
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};

int main()
{
    struct CPUID_1_EAX *tmp;
    int b[4];

#ifdef _MSC_VER
    __cpuid(b,1);
#endif

#ifdef __GNUC__
    cpuid (1, &b[0], &b[1], &b[2], &b[3]);
#endif

    tmp=(struct CPUID_1_EAX *)&b[0];
```

[162]Complément sur le fonctionnement interne de l'assembleur GCC

```
    printf ("stepping=%d\n", tmp->stepping);
    printf ("model=%d\n", tmp->model);
    printf ("family_id=%d\n", tmp->family_id);
    printf ("processor_type=%d\n", tmp->processor_type);
    printf ("extended_model_id=%d\n", tmp->extended_model_id);
    printf ("extended_family_id=%d\n", tmp->extended_family_id);

    return 0;
};
```

Après que l'instruction CPUID ait rempli les registres EAX/EBX/ECX/EDX, ceux-ci doivent être recopiés dans le tableau b[]. Nous affectons dont le pointeur de structure CPUID_1_EAX pour qu'il contienne l'adresse du tableau b[].

En d'autres termes, nous traitons une valeur *int* comme une structure, puis nous lisons des bits spécifiques de la structure.

MSVC

Compilons notre exemple avec MSVC 2008 en utilisant l'option /Ox :

Listing 1.353 : avec optimisation MSVC 2008

```
_b$ = -16  ; size = 16
_main    PROC
    sub    esp, 16
    push   ebx

    xor    ecx, ecx
    mov    eax, 1
    cpuid
    push   esi
    lea    esi, DWORD PTR _b$[esp+24]
    mov    DWORD PTR [esi], eax
    mov    DWORD PTR [esi+4], ebx
    mov    DWORD PTR [esi+8], ecx
    mov    DWORD PTR [esi+12], edx

    mov    esi, DWORD PTR _b$[esp+24]
    mov    eax, esi
    and    eax, 15
    push   eax
    push   OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
    call   _printf

    mov    ecx, esi
    shr    ecx, 4
    and    ecx, 15
    push   ecx
    push   OFFSET $SG15436 ; 'model=%d', 0aH, 00H
    call   _printf
```

```
    mov     edx, esi
    shr     edx, 8
    and     edx, 15
    push    edx
    push    OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
    call    _printf

    mov     eax, esi
    shr     eax, 12
    and     eax, 3
    push    eax
    push    OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
    call    _printf

    mov     ecx, esi
    shr     ecx, 16
    and     ecx, 15
    push    ecx
    push    OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
    call    _printf

    shr     esi, 20
    and     esi, 255
    push    esi
    push    OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
    call    _printf
    add     esp, 48
    pop     esi

    xor     eax, eax
    pop     ebx

    add     esp, 16
    ret     0
_main    ENDP
```

L'instruction SHR va décaler la valeur du registre EAX d'un certain nombre de bits qui vont être abandonnées. Nous ignorons donc certains des bits de la partie droite.

L'instruction AND "efface" les bits inutiles sur la gauche, ou en d'autres termes, ne laisse dans le registre EAX que les bits qui nous intéressent.

MSVC + OllyDbg

Chargeons notre exemple dans OllyDbg et voyons quelles valeurs sont présentes dans EAX/EBX/ECX/EDX après exécution de l'instruction CPUID:

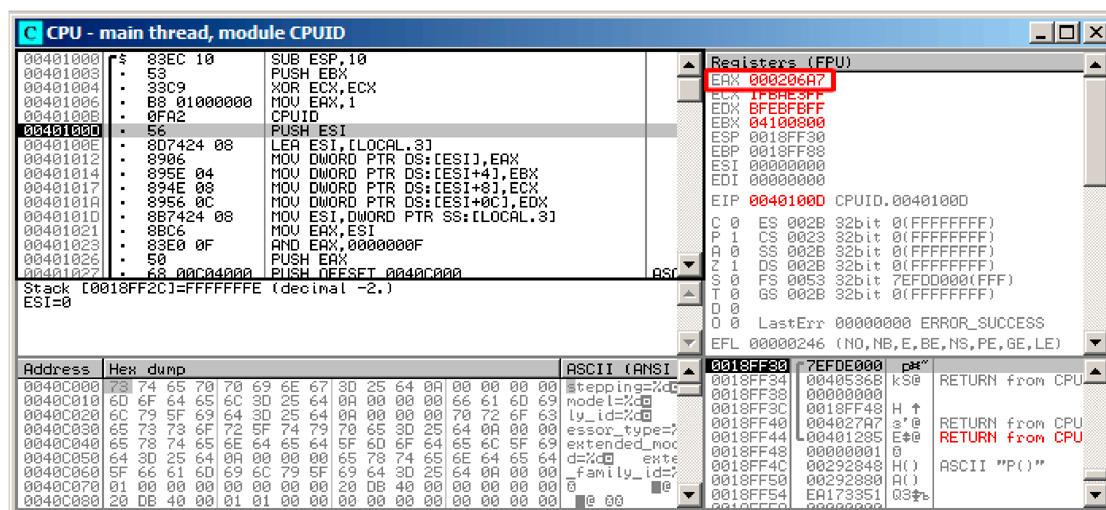


Fig. 1.108: OllyDbg : Après exécution de CPUID

La valeur de EAX est `0x000206A7` (ma CPU est un Intel Xeon E3-1220).
Cette valeur exprimée en binaire vaut $0b00000000000000100000011010100111$.

Voici la manière dont les bits sont répartis sur les différents champs:

champ	format binaire	format décimal
reserved2	0000	0
extended_family_id	00000000	0
extended_model_id	0010	2
reserved1	00	0
processor_id	00	0
family_id	0110	6
model	1010	10
stepping	0111	7

Listing 1.354 : Console output

```
stepping=7
model=10
family_id=6
processor_type=0
extended_model_id=2
extended_family_id=0
```

GCC

Essayons maintenant une compilation avec GCC 4.4.1 en utilisant l'option -03.

Listing 1.355 : avec optimisation GCC 4.4.1

```
main            proc near ; DATA XREF: _start+17
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h
    push    esi
    mov     esi, 1
    push    ebx
    mov     eax, esi
    sub     esp, 18h
    cpuid
    mov     esi, eax
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aSteppingD ; "stepping=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 4
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aModelD ; "model=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 8
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aFamily_idD ; "family_id=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 0Ch
    and     eax, 3
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aProcessor_type ; "processor_type=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 10h
    shr     esi, 14h
    and     eax, 0Fh
    and     esi, 0FFh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aExtended_model ;
    "extended_model_id=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     [esp+8], esi
```

```
    mov     dword ptr [esp+4], offset unk_80486D0
    mov     dword ptr [esp], 1
    call    ___printf_chk
    add     esp, 18h
    xor     eax, eax
    pop     ebx
    pop     esi
    mov     esp, ebp
    pop     ebp
    retn
main            endp
```

Le résultat est quasiment identique. Le seul élément notable est que GCC combine en quelques sortes le calcul de `extended_model_id` et `extended_family_id` en un seul bloc au lieu de les calculer séparément avant chaque appel à `printf()`.

Travailler avec le type float comme une structure

Comme nous l'avons expliqué dans la section traitant de la FPU (1.25 on page 285), les types *float* et *double* sont constitués d'un *signe*, d'un *significande* (ou *fraction*) et d'un *exposant*. Mais serions nous capable de travailler avec chacun de ces champs indépendamment? Essayons avec un *float*.

(S — signe)

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction : 23; // fraction
    unsigned int exponent : 8;  // exposant + 0x3FF
    unsigned int sign : 1;      // bit de signe
};

float f(float _in)
{
    float f=_in;
    struct float_as_struct t;

    assert (sizeof (struct float_as_struct) == sizeof (float));

    memcpy (&t, &f, sizeof (float));

    t.sign=1; // Positionnons le bit de signe
    t.exponent=t.exponent+2; // multiplions d par 2^n(n vaut 2 ici)
```

```
    memcpy (&f, &t, sizeof (float));

    return f;
};

int main()
{
    printf ("%f\n", f(1.234));
};
```

La structure `float_as_struct` occupe le même espace qu'un *float*, soit 4 octets ou 32 bits.

Nous positionnons maintenant le signe pour qu'il soit négatif puis en ajoutant à la valeur de l'exposant, ce qui fait que nous multiplions le nombre par 2^2, soit 4.

Compilons notre exemple avec MSVC 2008, sans optimisation:

Listing 1.356 : MSVC 2008 sans optimisation

```
_t$ = -8   ; size = 4
_f$ = -4   ; size = 4
__in$ = 8  ; size = 4
?f@@YAMM@Z PROC ; f
    push   ebp
    mov    ebp, esp
    sub    esp, 8

    fld    DWORD PTR __in$[ebp]
    fstp   DWORD PTR _f$[ebp]

    push   4
    lea    eax, DWORD PTR _f$[ebp]
    push   eax
    lea    ecx, DWORD PTR _t$[ebp]
    push   ecx
    call   _memcpy
    add    esp, 12

    mov    edx, DWORD PTR _t$[ebp]
    or     edx, -2147483648 ; 80000000H - positionnement du site négatif
    mov    DWORD PTR _t$[ebp], edx

    mov    eax, DWORD PTR _t$[ebp]
    shr    eax, 23          ; 00000017H - suppression du signifiant
    and    eax, 255         ; 000000ffH - nous ne conservons ici que
    l'exposant
    add    eax, 2           ; ajouter 2
    and    eax, 255         ; 000000ffH
    shl    eax, 23          ; 00000017H - décalage du résultat pour supprimer
    les bits 30:23
    mov    ecx, DWORD PTR _t$[ebp]
    and    ecx, -2139095041 ; 807fffffH - suppression de l'exposant
```

```
; ajout de la valeur originale de l'exposant avec le nouvel exposant qui
    vient d'être calculé:
    or      ecx, eax
    mov     DWORD PTR _t$[ebp], ecx

    push    4
    lea     edx, DWORD PTR _t$[ebp]
    push    edx
    lea     eax, DWORD PTR _f$[ebp]
    push    eax
    call    _memcpy
    add     esp, 12

    fld     DWORD PTR _f$[ebp]

    mov     esp, ebp
    pop     ebp
    ret     0
?f@@YAMM@Z ENDP    ; f
```

Si nous avions compilé avec le flag /Ox il n'y aurait pas d'appel à la fonction memcpy(), et la variable f serait utilisée directement. Mais la compréhension est facilitée lorsque l'on s'intéresse à la version non optimisée.

A quoi cela ressemblerait si nous utilisions l'option -O3 avec le compilateur GCC 4.4.1 ?

Listing 1.357 : GCC 4.4.1 avec optimisation

```
; f(float)
       public _Z1ff
_Z1ff  proc near

var_4  = dword ptr -4
arg_0  = dword ptr  8

       push    ebp
       mov     ebp, esp
       sub     esp, 4
       mov     eax, [ebp+arg_0]
       or      eax, 80000000h  ; positionnement du signe négatif
       mov     edx, eax
       and     eax, 807FFFFFh  ; Nous ne conservons que le signe et le
    signifiant dans EAX
       shr     edx, 23         ; Préparation de l'exposant
       add     edx, 2          ; Ajout de 2
       movzx   edx, dl         ; RAZ de tous les octets dans EDX à
    l'exception des bits 7:0
       shl     edx, 23         ; Décalage du nouvel exposant pour qu'ils soit
    à sa place
       or      eax, edx        ; Consolidation du nouvel exposant et de la
    valeur originale de l'exposant
       mov     [ebp+var_4], eax
       fld     [ebp+var_4]
       leave
```

```
        retn
_Z1ff   endp

        public main
main    proc near
        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 10h
        fld     ds:dword_8048614 ; -4.936
        fstp    qword ptr [esp+8]
        mov     dword ptr [esp+4], offset asc_8048610 ; "%f\n"
        mov     dword ptr [esp], 1
        call    ___printf_chk
        xor     eax, eax
        leave
        retn
main    endp
```

La fonction `f()` est à peu près compréhensible. Par contre ce qui est intéressant c'est que GCC a été capable de calculer le résultat de `f(1.234)` durant la compilation malgré tous les triturages des champs de la structure et a directement préparé l'argument passé à `printf()` durant la compilation!

1.30.7 Exercices

- http://challenges.re/71
- http://challenges.re/72

1.31 Le bogue *struct* classique

Ceci est un bogue *struct* classique.

Voici un exemple de définition:

```
struct test
{
        int field1;
        int field2;
};
```

Et puis les fichiers C:

```
void setter(struct test *t, int a, int b)
{
        t->field1=a;
        t->field2=b;
};
```

```
#include <stdio.h>

void printer(struct test *t)
{
        printf ("%d\n", t->field1);
        printf ("%d\n", t->field2);
};
```

Jusqu'ici, tout va bien.

Maintenant vous ajoutez un troisième champ dans la structure, entre les deux champs:

```
struct test
{
        int field1;
        int inserted;
        int field2;
};
```

Et vous modifiez probablement la fonction `setter()`, mais oubliez `printer()` :

```
void setter(struct test *t, int a, int b, int c)
{
        t->field1=a;
        t->inserted=b;
        t->field2=c;
};
```

Vous compilez votre projet, mais le fichier C où se trouve `printer()` qui est séparé, n'est pas recompilé car votre IDE[163] ou système de compilation n'a pas d'idée que ce module dépend d'une définition de structure *test*. Peut-être car `#include <new.h>` est oublié. Ou peut-être que le fichier d'entête `new.h` est inclus dans `printer.c` via un autre fichier d'entête. Le fichier objet n'est pas modifié (l'IDE pense qu'il n'a pas besoin d'être recompilé), tandis que la fonction `setter()` est déjà la nouvelle version. Ces deux fichiers objets (ancien et nouveau) peuvent tôt ou tard être liés dans un fichier exécutable.

Ensuite, vous le lancez, et le `setter()` met les 3 champs aux offsets +0, +4 et +8. Toutefois. `printer()` connait seulement 2 champs, et les prends aux offsets +0 et +4 lors de l'affichage.

Ceci conduits à des bogues obscurs et méchants. La raison est que l'IDE ou le système de construction ou le Makefile ne savent pas que les deux fichiers C (ou modules) dépendent de l'entête. Un remède courant est de tout supprimer et de recompiler.

Ceci est également vrai pour les classes C++, puisqu'elles fonctionnent tout comme des structures: 3.21.1 on page 698.

Ceci est une maladie ce C/C++, et une source de critique, oui. De nombreux LPs ont un meilleur support des modules et interfaces. Mais gardez à l'esprit l'époque de

[163] Integrated development environment

création du ocmpilateur C: dans les années 70, sur de vieux ordinateurs PDP. Donc tout a été simplifié à ceci par les créateurs du C.

1.32 Unions

Les *unions* en C/C++ sont utilisées principalement pour interpréter une variable (ou un bloc de mémoire) d'un type de données comme une variable d'un autre type de données.

1.32.1 Exemple de générateur de nombres pseudo-aléatoires

Si nous avons besoin de nombres aléatoires à virgule flottante entre 0 et 1, le plus simple est d'utiliser un PRNG comme le Twister de Mersenne. Il produit une valeur aléatoire non signée sur 32-bit (en d'autres mots, il produit une valeur 32-bit aléatoire). Puis, nous pouvons transformer cette valeur en *float* et le diviser par `RAND_MAX` (`0xFFFFFFFF` dans notre cas)—nous obtenons une valeur dans l'intervalle 0..1.

Mais nous savons que la division est lente. Aussi, nous aimerions utiliser le moins d'opérations FPU possible. Peut-on se passer de la division?

Rappelons-nous en quoi consiste un nombre en virgule flottante: un bit de signe, un significande et un exposant. Nous n'avons qu'à stocker des bits aléatoires dans toute le significande pour obtenir un nombre réel aléatoire!

L'exposant ne peut pas être zéro (le nombre flottant est dénormalisé dans ce cas), donc nous stockons 0b01111111 dans l'exposant—ce qui signifie que l'exposant est 1. Ensuite nous remplissons le significande avec des bits aléatoires, mettons le signe à 0 (ce qui indique un nombre positif) et voilà. Les nombres générés sont entre 1 et 2, donc nous devons soustraire 1.

Un générateur congruentiel linéaire de nombres aléatoire très simple est utilisé dans mon exemple[164], il produit des nombres 32-bit. Le PRNG est initialisé avec le temps courant au format UNIX timestamp.

Ici, nous représentons un type *float* comme une *union*—c'est la construction C/C++ qui nous permet d'interpréter un bloc de mémoire sous différents types. Dans notre cas, nous pouvons créer une variable de type *union* et y accéder comme si c'est un *float* ou un *uint32_t*. On peut dire que c'est juste un hack. Un sale.

Le code du PRNG entier est le même que celui que nous avons déjà considéré: 1.29 on page 436. Donc la forme compilée du code est omise.

```
#include <stdio.h>
#include <stdint.h>
#include <time.h>

// PRNG entier définitions, données et routines:

// constantes provenant du livre Numerical Recipes
const uint32_t RNG_a=1664525;
```

[164] l'idée a été prise de: URL

```
const uint32_t RNG_c=1013904223;
uint32_t RNG_state; // variable globale

void my_srand(uint32_t i)
{
        RNG_state=i;
};

uint32_t my_rand()
{
        RNG_state=RNG_state*RNG_a+RNG_c;
        return RNG_state;
};

// PRNG FPU définitions et routines:

union uint32_t_float
{
        uint32_t i;
        float f;
};

float float_rand()
{
        union uint32_t_float tmp;
        tmp.i=my_rand() & 0x007fffff | 0x3F800000;
        return tmp.f-1;
};

// test

int main()
{
        my_srand(time(NULL)); // initialisation du PRNG

        for (int i=0; i<100; i++)
                printf ("%f\n", float_rand());

        return 0;
};
```

x86

Listing 1.358 : MSVC 2010 avec optimisation

```
$SG4238 DB      '%f', 0aH, 00H

__real@3ff0000000000000 DQ 03ff0000000000000r   ; 1

tv130 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
        push    ecx
```

```
        call    ?my_rand@@YAIXZ
; EAX=valeur pseudo-aléatoire
        and     eax, 8388607            ; 007fffffH
        or      eax, 1065353216         ; 3f800000H
; EAX=valeur pseudo-aléatoire & 0x007fffff | 0x3f800000
; la stocker dans la pile locale:
        mov     DWORD PTR _tmp$[esp+4], eax
; la recharger en tant que nombre à virgule flottante:
        fld     DWORD PTR _tmp$[esp+4]
; soustraire 1.0:
        fsub    QWORD PTR __real@3ff0000000000000
; stocker la valeur obtenue dans la pile locale et la recharger:
; ces instructions sont redondantes:
        fstp    DWORD PTR tv130[esp+4]
        fld     DWORD PTR tv130[esp+4]
        pop     ecx
        ret     0
?float_rand@@YAMXZ ENDP

_main   PROC
        push    esi
        xor     eax, eax
        call    _time
        push    eax
        call    ?my_srand@@YAXI@Z
        add     esp, 4
        mov     esi, 100
$LL3@main:
        call    ?float_rand@@YAMXZ
        sub     esp, 8
        fstp    QWORD PTR [esp]
        push    OFFSET $SG4238
        call    _printf
        add     esp, 12
        dec     esi
        jne     SHORT $LL3@main
        xor     eax, eax
        pop     esi
        ret     0
_main   ENDP
```

Les noms de fonctions sont étranges ici car cet exemple a été compilé en tant que C++ et ceci est la modification des noms en C++, nous en parlerons plus loin: 3.21.1 on page 699. Si nous compilons ceci avec MSVC 2012, il utilise des instructions SIMD pour le FPU, pour en savoir plus: 1.38.5 on page 567.

ARM (Mode ARM)

Listing 1.359 : GCC 4.6.3 avec optimisation (IDA)

```
float_rand
                STMFD   SP!, {R3,LR}
                BL      my_rand
```

```
; R0=valeur pseudo-aléatoire
                FLDS    S0, =1.0
; S0=1.0
                BIC     R3, R0, #0xFF000000
                BIC     R3, R3, #0x800000
                ORR     R3, R3, #0x3F800000
; R3=valeur pseudo-aléatoire & 0x007fffff | 0x3f800000
; copier de R3 vers FPU (registre S15).
; ça se comporte comme une copie bit à bit, pas de conversion faite:
                FMSR    S15, R3
; soustraire 1.0 et laisser le résultat dans S0:
                FSUBS   S0, S15, S0
                LDMFD   SP!, {R3,PC}

flt_5C          DCFS 1.0

main
                STMFD   SP!, {R4,LR}
                MOV     R0, #0
                BL      time
                BL      my_srand
                MOV     R4, #0x64 ; 'd'

loc_78
                BL      float_rand
; S0=valeur pseudo-aléatoire
                LDR     R0, =aF         ; "%f"
; convertir la valeur obtenue en type double (printf() en a besoin):
                FCVTDS  D7, S0
; copie bit à bit de D7 dans la paire de registres R2/R3 (pour printf()):
                FMRRD   R2, R3, D7
                BL      printf
                SUBS    R4, R4, #1
                BNE     loc_78
                MOV     R0, R4
                LDMFD   SP!, {R4,PC}

aF              DCB "%f",0xA,0
```

Nous allons faire un dump avec objdump et nous allons voir que les instructions FPU ont un nom différent que dans IDA. Apparemment, les développeurs de IDA et binutils ont utilisés des manuels différents? Peut-être qu'il serait bon de connaître les deux variantes de noms des instructions.

Listing 1.360 : GCC 4.6.3 avec optimisation (objdump)

```
00000038 <float_rand>:
  38:   e92d4008        push    {r3, lr}
  3c:   ebfffffe        bl      10 <my_rand>
  40:   ed9f0a05        vldr    s0, [pc, #20]   ; 5c <float_rand+0x24>
  44:   e3c034ff        bic     r3, r0, #-16777216      ; 0xff000000
  48:   e3c33502        bic     r3, r3, #8388608        ; 0x800000
  4c:   e38335fe        orr     r3, r3, #1065353216     ; 0x3f800000
  50:   ee073a90        vmov    s15, r3
```

```
  54:   ee370ac0        vsub.f32        s0, s15, s0
  58:   e8bd8008        pop     {r3, pc}
  5c:   3f800000        svccc   0x00800000

00000000 <main>:
   0:   e92d4010        push    {r4, lr}
   4:   e3a00000        mov     r0, #0
   8:   ebfffffe        bl      0 <time>
   c:   ebfffffe        bl      0 <main>
  10:   e3a04064        mov     r4, #100        ; 0x64
  14:   ebfffffe        bl      38 <main+0x38>
  18:   e59f0018        ldr     r0, [pc, #24]   ; 38 <main+0x38>
  1c:   eeb77ac0        vcvt.f64.f32    d7, s0
  20:   ec532b17        vmov    r2, r3, d7
  24:   ebfffffe        bl      0 <printf>
  28:   e2544001        subs    r4, r4, #1
  2c:   1afffff8        bne     14 <main+0x14>
  30:   e1a00004        mov     r0, r4
  34:   e8bd8010        pop     {r4, pc}
  38:   00000000        andeq   r0, r0, r0
```

Les instructions en 0x5c dans `float_rand()` et en 0x38 dans `main()` sont du bruit (pseudo-)aléatoire.

1.32.2 Calcul de l'epsilon de la machine

L'epsilon de la machine est la plus petite valeur avec laquelle le FPU peut travailler. Plus il y a de bits alloués pour représenter un nombre en virgule flottante, plus l'epsilon est petit, C'est $2^{-23} \approx 1.19e-07$ pour les *float* et $2^{-52} \approx 2.22e-16$ pour les *double*. Voir aussi: l'article de Wikipédia.

Il intéressant de voir comme il est facile de calculer l'epsilon de la machine:

```
#include <stdio.h>
#include <stdint.h>

union uint_float
{
        uint32_t i;
        float f;
};

float calculate_machine_epsilon(float start)
{
        union uint_float v;
        v.f=start;
        v.i++;
        return v.f-start;
}

void main()
{
        printf ("%g\n", calculate_machine_epsilon(1.0));
```

```
};
```

Ce que l'on fait ici est simplement de traiter la partie fractionnaire du nombre au format IEEE 754 comme un entier et de lui ajouter 1. Le nombre flottant en résultant est égal à $starting_value + machine_epsilon$, donc il suffit de soustraire $starting_value$ (en utilisant l'arithmétique flottante) pour mesurer ce que la différence d'un bit représente dans un nombre flottant simple précision(*float*). L' *union* permet ici d'accéder au nombre IEEE 754 comme à un entier normal. Lui ajouter 1 ajoute en fait 1 au *significande* du nombre, toutefois, inutile de dire, un débordement est possible, qui ajouterait 1 à l'exposant.

x86

Listing 1.361 : avec optimisation MSVC 2010

```
tv130 = 8
_v$ = 8
_start$ = 8
_calculate_machine_epsilon PROC
        fld     DWORD PTR _start$[esp-4]
; cette instruction est redondante:
        fst     DWORD PTR _v$[esp-4]
        inc     DWORD PTR _v$[esp-4]
        fsubr   DWORD PTR _v$[esp-4]
; cette paire d'instructions est aussi redondante:
        fstp    DWORD PTR tv130[esp-4]
        fld     DWORD PTR tv130[esp-4]
        ret     0
_calculate_machine_epsilon ENDP
```

La seconde instruction FST est redondante: il n'est pas nécessaire de stocker la valeur en entrée à la même place (le compilateur a décidé d'allouer la variable v à la même place dans la pile locale que l'argument en entrée). Puis elle est incrémentée avec INC, puisque c'est une variable entière normale. Ensuite elle est chargée dans le FPU comme un nombre IEEE 754 32-bit, FSUBR fait le reste du travail et la valeur résultante est stockée dans ST0. La dernière paire d'instructions FSTP/FLD est redondante, mais le compilateur n'a pas optimisé le code.

ARM64

Étendons notre exemple à 64-bit:

```
#include <stdio.h>
#include <stdint.h>

typedef union
{
        uint64_t i;
        double d;
} uint_double;

```

```
double calculate_machine_epsilon(double start)
{
        uint_double v;
        v.d=start;
        v.i++;
        return v.d-start;
}

void main()
{
        printf ("%g\n", calculate_machine_epsilon(1.0));
};
```

ARM64 n'a pas d'instruction qui peut ajouter un nombre a un D-registre FPU, donc la valeur en entrée (qui provient du registre x64 D0) est d'abord copiée dans le GPR, incrémentée, copiée dans le registre FPU D1, et puis la soustraction est faite.

Listing 1.362 : GCC 4.9 ARM64 avec optimisation

```
calculate_machine_epsilon:
        fmov    x0, d0          ; charger la valeur d'entrée de type double dans
    X0
        add     x0, x0, 1       ; X0++
        fmov    d1, x0          ; la déplacer dans le registre du FPU
        fsub    d0, d1, d0      ; soustraire
        ret
```

Voir aussi cet exemple compilé pour x64 avec instructions SIMD: 1.38.4 on page 566.

MIPS

Il y a ici la nouvelle instruction MTC1 («Move To Coprocessor 1 »), elle transfère simplement des données vers les registres du FPU.

Listing 1.363 : GCC 4.4.5 avec optimisation (IDA)

```
calculate_machine_epsilon:
                mfc1    $v0, $f12
                or      $at, $zero ; NOP
                addiu   $v1, $v0, 1
                mtc1    $v1, $f2
                jr      $ra
                sub.s   $f0, $f2, $f12 ; branch delay slot
```

Conclusion

Il est difficile de dire si quelqu'un pourrait avoir besoin de cette astuce dans du code réel, mais comme cela a été mentionné plusieurs fois dans ce livre, cet exemple est utile pour expliquer le format IEEE 754 et les *union*s en C/C++.

1.32.3 Remplacement de FSCALE

Agner Fog dans son travail[165] *Optimizing subroutines in assembly language / An optimization guide for x86 platforms* indique que l'instruction FPU FSCALE (qui calcule 2^n) peut être lente sur de nombreux CPUs, et propose un remplacement plus rapide.

Voici ma conversion de son code assembleur en C/C++ :

```
#include <stdint.h>
#include <stdio.h>

union uint_float
{
        uint32_t i;
        float f;
};

float flt_2n(int N)
{
        union uint_float tmp;

        tmp.i=(N<<23)+0x3f800000;
        return tmp.f;
};

struct float_as_struct
{
        unsigned int fraction : 23;
        unsigned int exponent : 8;
        unsigned int sign : 1;
};

float flt_2n_v2(int N)
{
        struct float_as_struct tmp;

        tmp.fraction=0;
        tmp.sign=0;
        tmp.exponent=N+0x7f;
        return *(float*)(&tmp);
};

union uint64_double
{
        uint64_t i;
        double d;
};

double dbl_2n(int N)
{
        union uint64_double tmp;

        tmp.i=((uint64_t)N<<52)+0x3ff0000000000000UL;
```

165. http://www.agner.org/optimize/optimizing_assembly.pdf

```
        return tmp.d;
};

struct double_as_struct
{
        uint64_t fraction : 52;
        int exponent : 11;
        int sign : 1;
};

double dbl_2n_v2(int N)
{
        struct double_as_struct tmp;

        tmp.fraction=0;
        tmp.sign=0;
        tmp.exponent=N+0x3ff;
        return *(double*)(&tmp);
};

int main()
{
        // 2^11 = 2048
        printf ("%f\n", flt_2n(11));
        printf ("%f\n", flt_2n_v2(11));
        printf ("%lf\n", dbl_2n(11));
        printf ("%lf\n", dbl_2n_v2(11));
};
```

L'instruction FSCALE peut être plus rapide dans votre environnement, mais néanmoins, c'est un bon exemple d'*union* et du fait que l'exposant est stocké sous la forme 2^n, donc une valeur n en entrée est décalée à l'exposant dans le nombre encodé en IEEE 754. Ensuite, l'exposant est corrigé avec l'ajout de 0x3f800000 ou de 0x3ff0000000000000.

La même chose peut être faite sans décalage utilisant *struct*, mais en interne, l'opération de décalage aura toujours lieu.

1.32.4 Calcul rapide de racine carré

Un autre algorithme connu où un *float* est interprété comme un entier est celui de calcul rapide de racine carrée.

Listing 1.364 : Le code source provient de Wikipédia: https://en.wikipedia.org/wiki/Methods_of_computing_square_roots

```
/* Assumes that float is in the IEEE 754 single precision floating point ↙
    ↘ format
 * and that int is 32 bits. */
float sqrt_approx(float z)
{
    int val_int = *(int*)&z; /* Same bits, but as an int */
    /*
```

```
    * To justify the following code, prove that
    *
    * ((((val_int / 2^m) - b) / 2) + b) * 2^m = ((val_int - 2^m) / 2) + ((
    b + 1) / 2) * 2^m)
    *
    * where
    *
    * b = exponent bias
    * m = number of mantissa bits
    *
    * .
    */

   val_int -= 1 << 23; /* Subtract 2^m. */
   val_int >>= 1; /* Divide by 2. */
   val_int += 1 << 29; /* Add ((b + 1) / 2) * 2^m. */

   return *(float*)&val_int; /* Interpret again as float */
}
```

À titre d'exercice, vous pouvez essayez de compiler cette fonction et de comprendre comme elle fonctionne.

C'est un algorithme connu de calcul rapide de $\frac{1}{\sqrt{x}}$. L'algorithme devînt connu, supposément, car il a été utilisé dans Quake III Arena.

La description de l'algorithme peut être trouvée sur Wikipédia: http://en.wikipedia.org/wiki/Fast_inverse_square_root.

1.33 Pointeurs sur des fonctions

Un pointeur sur une fonction, comme tout autre pointeur, est juste l'adresse de début de la fonction dans son segment de code.

Ils sont souvent utilisés pour appeler des fonctions callback (de rappel).

Des exemples bien connus sont:

- `qsort()`, `atexit()` de la bibliothèque C standard;
- signaux des OS *NIX;
- démarrage de thread: `CreateThread()` (win32), `pthread_create()` (POSIX);
- beaucoup de fonctions win32, comme `EnumChildWindows()`.
- dans de nombreux endroits du noyau Linux, par exemple les fonctions des drivers du système de fichier sont appelées via callbacks.
- Les fonctions des plugins GCC sont aussi appelées via callbacks.

Donc, la fonction `qsort()` est une implémentation du tri rapide dans la bibliothèque standard C/C++. La fonction est capable de trier n'importe quoi, tout type de don-

nées, tant que vous avez une fonction pour comparer deux éléments et que qsort() est capable de l'appeler.

La fonction de comparaison peut être définie comme:

```
int (*compare)(const void *, const void *)
```

Utilisons l'exemple suivant:

```
/* ex3 Sorting ints with qsort */

#include <stdio.h>
#include <stdlib.h>

int comp(const void * _a, const void * _b)
{
  const int *a=(const int *)_a;
  const int *b=(const int *)_b;

  if (*a==*b)
    return 0;
  else
    if (*a < *b)
        return -1;
     else
      return 1;
}

int main(int argc, char* argv[])
{
   int numbers[10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
   int i;

  /* Sort the array */
  qsort(numbers,10,sizeof(int),comp) ;
  for (i=0;i<9;i++)
    printf("Number = %d\n",numbers[ i ]) ;
  return 0;
}
```

1.33.1 MSVC

Compilons le dans MSVC 2010 (certaines parties ont été omises, dans un but de concision) avec l'option /Ox :

Listing 1.365 : MSVC 2010 avec optimisation : /GS- /MD

```
__a$ = 8                                                   ; size = 4
__b$ = 12                                                  ; size = 4
_comp   PROC
        mov     eax, DWORD PTR __a$[esp-4]
        mov     ecx, DWORD PTR __b$[esp-4]
        mov     eax, DWORD PTR [eax]
```

```
        mov     ecx, DWORD PTR [ecx]
        cmp     eax, ecx
        jne     SHORT $LN4@comp
        xor     eax, eax
        ret     0
$LN4@comp:
        xor     edx, edx
        cmp     eax, ecx
        setge   dl
        lea     eax, DWORD PTR [edx+edx-1]
        ret     0
_comp   ENDP


_numbers$ = -40                                         ; size = 40
_argc$ = 8                                              ; size = 4
_argv$ = 12                                             ; size = 4
_main   PROC
        sub     esp, 40                                 ; 00000028H
        push    esi
        push    OFFSET _comp
        push    4
        lea     eax, DWORD PTR _numbers$[esp+52]
        push    10                                      ; 0000000aH
        push    eax
        mov     DWORD PTR _numbers$[esp+60], 1892       ; 00000764H
        mov     DWORD PTR _numbers$[esp+64], 45         ; 0000002dH
        mov     DWORD PTR _numbers$[esp+68], 200        ; 000000c8H
        mov     DWORD PTR _numbers$[esp+72], -98        ; ffffff9eH
        mov     DWORD PTR _numbers$[esp+76], 4087       ; 00000ff7H
        mov     DWORD PTR _numbers$[esp+80], 5
        mov     DWORD PTR _numbers$[esp+84], -12345     ; ffffcfc7H
        mov     DWORD PTR _numbers$[esp+88], 1087       ; 0000043fH
        mov     DWORD PTR _numbers$[esp+92], 88         ; 00000058H
        mov     DWORD PTR _numbers$[esp+96], -100000    ; fffe7960H
        call    _qsort
        add     esp, 16                                 ; 00000010H

...
```

Rien de surprenant jusqu'ici. Comme quatrième argument, l'adresse du label _comp est passée, qui est juste l'endroit où se trouve comp(), ou, en d'autres mots, l'adresse de la première instruction de cette fonction.

Comment est-ce que qsort() l'appelle?

Regardons cette fonction, située dans MSVCR80.DLL (un module DLL de MSVC avec des fonctions de la bibliothèque C standard) :

Listing 1.366 : MSVCR80.DLL

```
.text:7816CBF0 ; void __cdecl qsort(void *, unsigned int, unsigned int, int
    (__cdecl *)(const void *, const void *))
.text:7816CBF0                  public _qsort
.text:7816CBF0 _qsort           proc near
```

```
.text:7816CBF0
.text:7816CBF0 lo              = dword ptr -104h
.text:7816CBF0 hi              = dword ptr -100h
.text:7816CBF0 var_FC          = dword ptr -0FCh
.text:7816CBF0 stkptr          = dword ptr -0F8h
.text:7816CBF0 lostk           = dword ptr -0F4h
.text:7816CBF0 histk           = dword ptr -7Ch
.text:7816CBF0 base            = dword ptr  4
.text:7816CBF0 num             = dword ptr  8
.text:7816CBF0 width           = dword ptr  0Ch
.text:7816CBF0 comp            = dword ptr  10h
.text:7816CBF0
.text:7816CBF0                 sub     esp, 100h

....

.text:7816CCE0 loc_7816CCE0:                           ; CODE XREF: _qsort+B1
.text:7816CCE0                 shr     eax, 1
.text:7816CCE2                 imul    eax, ebp
.text:7816CCE5                 add     eax, ebx
.text:7816CCE7                 mov     edi, eax
.text:7816CCE9                 push    edi
.text:7816CCEA                 push    ebx
.text:7816CCEB                 call    [esp+118h+comp]
.text:7816CCF2                 add     esp, 8
.text:7816CCF5                 test    eax, eax
.text:7816CCF7                 jle     short loc_7816CD04
```

comp—est le quatrième argument de la fonction. Ici, le contrôle est passé à l'adresse dans l'argument comp. Avant cela, deux arguments sont préparés pour comp(). Son résultat est testé après son exécution.

C'est pourquoi il est dangereux d'utiliser des pointeurs sur des fonctions. Tout d'abord, si vous appelez qsort() avec un pointeur de fonction incorrect, qsort() peut passer le contrôle du flux à un point incorrect, le processus peut planter et ce bug sera difficile à trouver.

La seconde raison est que les types de la fonction de callback doivent être strictement conforme, appeler la mauvaise fonction avec de mauvais arguments du mauvais type peut conduire à de sérieux problèmes, toutefois, le plantage du processus n'est pas un problème ici —le problème est de comment déterminer la raison du plantage —car le compilateur peut être silencieux sur le problème potentiel lors de la compilation.

MSVC + OllyDbg

Chargeons notre exemple dans OllyDbg et mettons un point d'arrêt sur `comp()`. Nous voyons comment les valeurs sont comparées lors du premier appel de `comp()` :

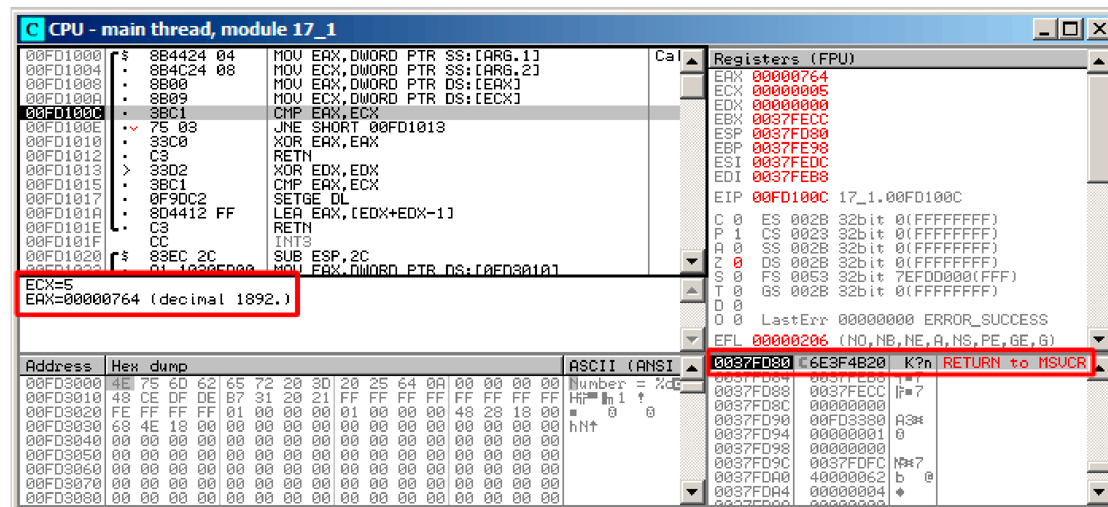


Fig. 1.109: OllyDbg : premier appel de `comp()`

OllyDbg montre les valeurs comparées dans la fenêtre sous celle du code, par commodité. Nous voyons que SP pointe sur RA, où se trouve la fonction `qsort()` (dans `MSVCR100.DLL`).

En traçant (F8) jusqu'à l'instruction RETN et appuyant sur F8 une fois de plus, nous retournons à la fonction `qsort()` :

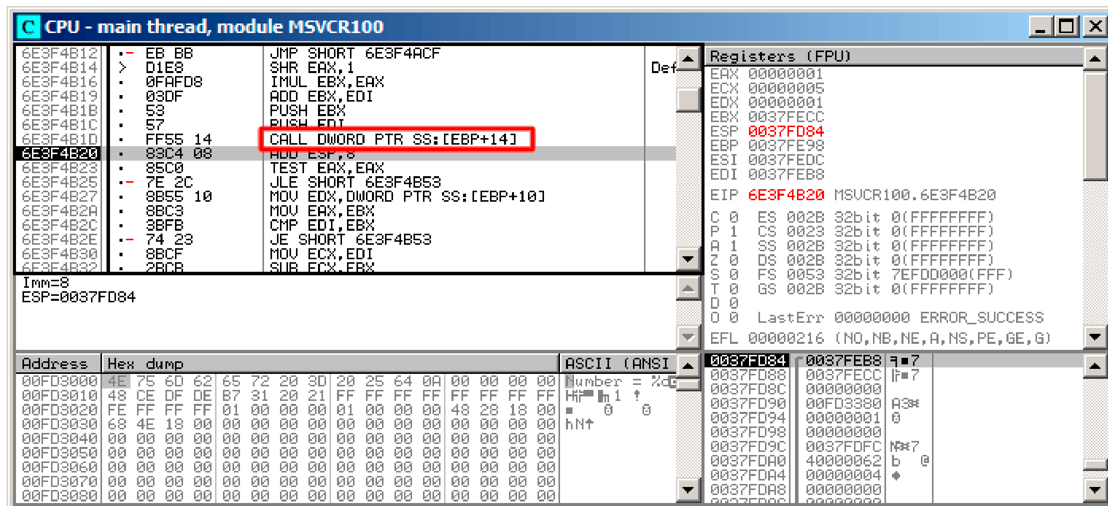


Fig. 1.110: OllyDbg : le code dans `qsort()` juste après l'appel de `comp()`

Ça a été un appel à la fonction de comparaison.

Voici aussi une copie d'écran au moment du second appel à`comp()`—maintenant les valeurs qui doivent être comparées sont différentes:

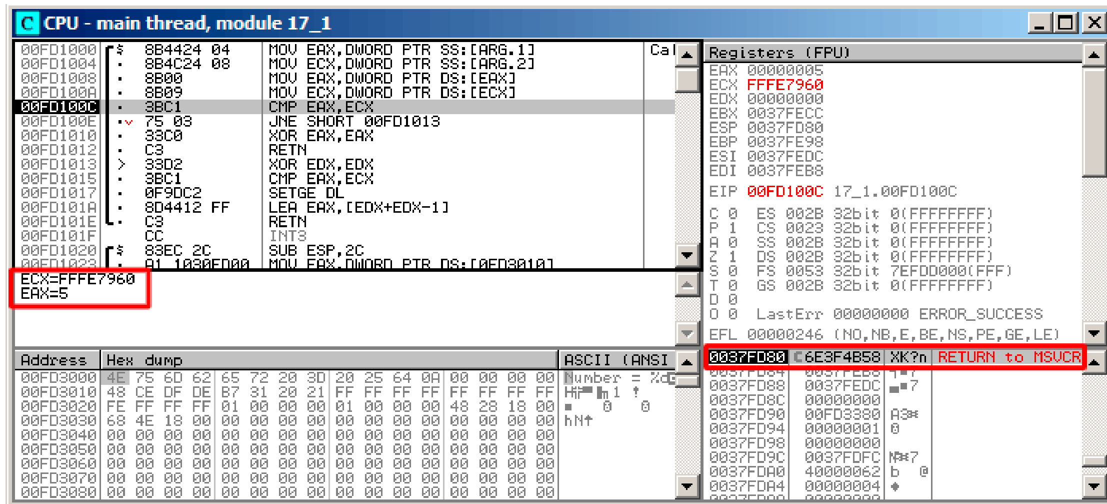


Fig. 1.111: OllyDbg : second appel de `comp()`

MSVC + tracer

Regardons quelles sont le paires comparées. Ces 10 nombres vont être triés: 1892, 45, 200, -98, 4087, 5, -12345, 1087, 88, -100000.

Nous avons l'adresse de la première instruction CMP dans `comp()`, c'est `0x0040100C` et nous y avons mis un point d'arrêt:

```
tracer.exe -l:17_1.exe bpx=17_1.exe!0x0040100C
```

Maintenant nous avons des informations sur les registres au point d'arrêt:

```
PID=4336|New process 17_1.exe
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xfffe7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=PF ZF IF
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=CF PF ZF IF
...
```

Filtrons sur EAX et ECX et nous obtenons:

```
EAX=0x00000764 ECX=0x00000005
EAX=0x00000005 ECX=0xfffe7960
EAX=0x00000764 ECX=0x00000005
EAX=0x0000002d ECX=0x00000005
EAX=0x00000058 ECX=0x00000005
EAX=0x0000043f ECX=0x00000005
EAX=0xffffcfc7 ECX=0x00000005
EAX=0x000000c8 ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x00000005 ECX=0xffffcfc7
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0xffffff9e ECX=0xffffcfc7
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x000000c8 ECX=0x00000ff7
EAX=0x0000002d ECX=0x00000ff7
EAX=0x0000043f ECX=0x00000ff7
EAX=0x00000058 ECX=0x00000ff7
EAX=0x00000764 ECX=0x00000ff7
EAX=0x000000c8 ECX=0x00000764
EAX=0x0000002d ECX=0x00000764
EAX=0x0000043f ECX=0x00000764
EAX=0x00000058 ECX=0x00000764
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000043f ECX=0x000000c8
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000002d ECX=0x00000058
```

Il y a 34 paires. C'est pourquoi, l'algorithme de tri rapide a besoin de 34 opérations de comparaison pour trier ces 10 nombres.

MSVC + tracer (couverture du code)

Nous pouvons aussi utiliser la capacité du tracer pour collecter tous les registres possible et les montrer dans IDA.

Exécutons pas à pas toutes les instructions dans `comp()` :

```
tracer.exe -l:17_1.exe bpf=17_1.exe!0x00401000,trace:cc
```

Nous obtenons un scipt .idc pour charger dans IDA et chargeons le:

```
.text:00401000
.text:00401000 ; int __cdecl PtFuncCompare(const void *, const void *)
.text:00401000 PtFuncCompare   proc near               ; DATA XREF: _main+5↓o
.text:00401000
.text:00401000 arg_0           = dword ptr  4
.text:00401000 arg_4           = dword ptr  8
.text:00401000
.text:00401000                 mov     eax, [esp+arg_0] ; [ESP+4]=0x45f7ec..0x45f810(step=4), L"?\x04?
.text:00401004                 mov     ecx, [esp+arg_4] ; [ESP+8]=0x45f7ec..0x45f7f4(step=4), 0x45f7fc
.text:00401008                 mov     eax, [eax]      ; [EAX]=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff
.text:0040100A                 mov     ecx, [ecx]      ; [ECX]=5, 0x58, 0xc8, 0x764, 0xff7, 0xfffe7960
.text:0040100C                 cmp     eax, ecx        ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:0040100E                 jnz     short loc_401013 ; ZF=false
.text:00401010                 xor     eax, eax
.text:00401012                 retn
.text:00401013 ; ---------------------------------------------------------------------------
.text:00401013
.text:00401013 loc_401013:                             ; CODE XREF: PtFuncCompare+E↑j
.text:00401013                 xor     edx, edx
.text:00401015                 cmp     eax, ecx        ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:00401017                 setnl   dl              ; SF=false,true OF=false
.text:0040101A                 lea     eax, [edx+edx-1]
.text:0040101E                 retn                    ; EAX=1, 0xffffffff
.text:0040101E PtFuncCompare   endp
.text:0040101F
```

Fig. 1.112: tracer et IDA. N.B.: certaines valeurs sont coupées à droite

IDA a donné un nom à la fonction (PtFuncCompare)—car IDA voit que le pointeur sur cette fonction est passé à `qsort()`.

Nous voyons que les pointeurs a et b pointent sur différents emplacements dans le tableau, mais le pas entre eux est 4, puisque des valeurs 32-bit sont stockées dans le tableau.

Nous voyons que les instructions en `0x401010` et `0x401012` ne sont jamais exécutées (donc elles sont laissées en blanc) : en effet, `comp()` ne renvoie jamais 0, car il n'y a pas d'éléments égaux dans le tableau.

1.33.2 GCC

Il n'y a pas beaucoup de différence:

Listing 1.367 : GCC

```
                lea     eax, [esp+40h+var_28]
                mov     [esp+40h+var_40], eax
```

```
                mov     [esp+40h+var_28], 764h
                mov     [esp+40h+var_24], 2Dh
                mov     [esp+40h+var_20], 0C8h
                mov     [esp+40h+var_1C], 0FFFFFF9Eh
                mov     [esp+40h+var_18], 0FF7h
                mov     [esp+40h+var_14], 5
                mov     [esp+40h+var_10], 0FFFFCFC7h
                mov     [esp+40h+var_C], 43Fh
                mov     [esp+40h+var_8], 58h
                mov     [esp+40h+var_4], 0FFFE7960h
                mov     [esp+40h+var_34], offset comp
                mov     [esp+40h+var_38], 4
                mov     [esp+40h+var_3C], 0Ah
                call    _qsort
```

Fonction comp() :

```
                public comp
comp            proc near

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_4]
                mov     ecx, [ebp+arg_0]
                mov     edx, [eax]
                xor     eax, eax
                cmp     [ecx], edx
                jnz     short loc_8048458
                pop     ebp
                retn
loc_8048458:
                setnl   al
                movzx   eax, al
                lea     eax, [eax+eax-1]
                pop     ebp
                retn
comp            endp
```

L'implémentation de qsort() se trouve dans libc.so.6 et c'est en fait juste un wrapper[166] pour qsort_r().

À son tour, elle appelle quicksort(), où notre fonction défini est appelée via le pointeur passé:

Listing 1.368 : (fihier libc.so.6, glibc version—2.10.1)

```
...
.text:0002DDF6                 mov     edx, [ebp+arg_10]
.text:0002DDF9                 mov     [esp+4], esi
```

[166]un concept similaire à une fonction thunk

```
.text:0002DDFD                 mov     [esp], edi
.text:0002DE00                 mov     [esp+8], edx
.text:0002DE04                 call    [ebp+arg_C]
...
```

GCC + GDB (avec code source)

Évidemment, nous avons le code source C de notre exemple (1.33 on page 496), donc nous pouvons mettre un point d'arrêt (b) sur le numéro de ligne (11—la ligne où la première comparaison se produit. Nous devons aussi compiler l'exemple avec les informations de débogage incluses (-g), donc la table avec les adresses et les numéros de ligne correspondants est présente.

Nous pouvons aussi afficher les valeurs en utilisant les noms de variable (p) : les informations de débogage nous disent aussi quel registre et/ou élément de la pile locale contient quelle variable.

Nous pouvons aussi voir la pile (bt) et y trouver qu'il y a une fonction intermédiaire msort_with_tmp() utilisée dans la Glibc.

Listing 1.369 : session GDB

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c -g
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...done.
(gdb) b 17_1.c:11
Breakpoint 1 at 0x804845f: file 17_1.c, line 11.
(gdb) run
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
11          if (*a==*b)
(gdb) p *a
$1 = 1892
(gdb) p *b
$2 = 45
(gdb) c
Continuing.

Breakpoint 1, comp (_a=0xbffff104, _b=_b@entry=0xbffff108) at 17_1.c:11
11          if (*a==*b)
(gdb) p *a
$3 = -98
(gdb) p *b
$4 = 4087
(gdb) bt
#0  comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0↙
    ↪ xbffff0f8, n=n@entry=2)
    at msort.c:65
```

```
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort⤦
    ↳ .c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry⤦
    ↳ =5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort⤦
    ↳ .c:45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry⤦
    ↳ =10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at ⤦
    ↳ msort.c:45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=⤦
    ↳ cmp@entry=0x804844d <comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp⤦
    ↳ >) at msort.c:307
#9  0x0804850d in main (argc=1, argv=0xbffff1c4) at 17_1.c:26
(gdb)
```

GCC + GDB (pas de code source)

Mais souvent, il n'y a pas de code source du tout, donc nous pouvons désassembler la fonction `comp()` (`disas`), trouver la toute première instruction `CMP` et placer un point d'arrêt (*b*) à cette adresse.

À chaque point d'arrêt, nous allons afficher le contenu de tous les registres (`info registers`). Le contenu de la pile est aussi disponible (`bt`),

mais partiellement: il n'y a pas l'information du numéro de ligne pour `comp()`.

Listing 1.370 : session GDB

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...(no debugging symbols ⤦
    ↳ found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas comp
Dump of assembler code for function comp:
   0x0804844d <+0>:     push   ebp
   0x0804844e <+1>:     mov    ebp,esp
   0x08048450 <+3>:     sub    esp,0x10
   0x08048453 <+6>:     mov    eax,DWORD PTR [ebp+0x8]
   0x08048456 <+9>:     mov    DWORD PTR [ebp-0x8],eax
   0x08048459 <+12>:    mov    eax,DWORD PTR [ebp+0xc]
   0x0804845c <+15>:    mov    DWORD PTR [ebp-0x4],eax
   0x0804845f <+18>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048462 <+21>:    mov    edx,DWORD PTR [eax]
   0x08048464 <+23>:    mov    eax,DWORD PTR [ebp-0x4]
   0x08048467 <+26>:    mov    eax,DWORD PTR [eax]
   0x08048469 <+28>:    cmp    edx,eax
```

```
   0x0804846b <+30>:    jne    0x8048474 <comp+39>
   0x0804846d <+32>:    mov    eax,0x0
   0x08048472 <+37>:    jmp    0x804848e <comp+65>
   0x08048474 <+39>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048477 <+42>:    mov    edx,DWORD PTR [eax]
   0x08048479 <+44>:    mov    eax,DWORD PTR [ebp-0x4]
   0x0804847c <+47>:    mov    eax,DWORD PTR [eax]
   0x0804847e <+49>:    cmp    edx,eax
   0x08048480 <+51>:    jge    0x8048489 <comp+60>
   0x08048482 <+53>:    mov    eax,0xffffffff
   0x08048487 <+58>:    jmp    0x804848e <comp+65>
   0x08048489 <+60>:    mov    eax,0x1
   0x0804848e <+65>:    leave
   0x0804848f <+66>:    ret
End of assembler dump.
(gdb) b *0x08048469
Breakpoint 1 at 0x8048469
(gdb) run
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0x2d     45
ecx            0xbffff0f8       -1073745672
edx            0x764    1892
ebx            0xb7fc0000       -1208221696
esp            0xbfffeeb8       0xbfffeeb8
ebp            0xbfffeec8       0xbfffeec8
esi            0xbffff0fc       -1073745668
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0xff7    4087
ecx            0xbffff104       -1073745660
edx            0xffffff9e       -98
ebx            0xb7fc0000       -1208221696
esp            0xbfffee58       0xbfffee58
ebp            0xbfffee68       0xbfffee68
esi            0xbffff108       -1073745656
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x282    [ SF IF ]
```

```
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0xffffff9e       -98
ecx            0xbffff100       -1073745664
edx            0xc8     200
ebx            0xb7fc0000       -1208221696
esp            0xbfffeeb8       0xbfffeeb8
ebp            0xbfffeec8       0xbfffeec8
esi            0xbffff104       -1073745660
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) bt
#0  0x08048469 in comp ()
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0⤦
    ↳ xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort⤦
    ↳ .c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry⤦
    ↳ =5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort⤦
    ↳ .c:45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry⤦
    ↳ =10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at ⤦
    ↳ msort.c:45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=⤦
    ↳ cmp@entry=0x804844d <comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp⤦
    ↳ >) at msort.c:307
#9  0x0804850d in main ()
```

1.33.3 Danger des pointeurs sur des fonctions

Comme nous pouvons le voir, la fonction `qsort()` attend un pointeur sur une fonction qui prend deux arguments de type *void** et renvoie un entier: Si vous avez plusieurs fonctions de comparaison dans votre code (une qui compare les chaînes, une autre—les entiers, etc.), il est très facile de les mélanger les unes avec les autres. Vous pouvez essayer de trier un tableau de chaîne en utilisant une fonction qui compare les entiers, et le compilateur ne vous avertira pas de ce bogue.

1.34 Valeurs 64-bit dans un environnement 32-bit

Dans un environnement 32-bit, les GPR sont 32-bit, donc les valeurs 64-bit sont stockées et passées comme une paire de registres 32-bit[167].

1.34.1 Renvoyer une valeur 64-bit

```
#include <stdint.h>

uint64_t f ()
{
        return 0x1234567890ABCDEF;
};
```

x86

Dans un environnement 32-bit, les valeurs 64-bit sont renvoyées des fonctions dans la paire de registres EDX :EAX.

Listing 1.371 : MSVC 2010 avec optimisation

```
_f      PROC
        mov     eax, -1867788817 ; 90abcdefH
        mov     edx, 305419896   ; 12345678H
        ret     0
_f      ENDP
```

ARM

Une valeur 64-bit est renvoyée dans la paire de registres `R0`-`R1` (`R1` est pour la partie haute et `R0` pour la partie basse) :

Listing 1.372 : avec optimisation Keil 6/2013 (Mode ARM)

```
||f|| PROC
        LDR      r0,|L0.12|
        LDR      r1,|L0.16|
        BX       lr
```

[167] A propos, les valeurs 32-bit sont passées en tant que paire dans les environnements 16-bit de la même manière: 3.34.4 on page 843

```
        ENDP

|L0.12|
        DCD      0x90abcdef
|L0.16|
        DCD      0x12345678
```

MIPS

Une valeur 64-bit est renvoyée dans la paire de registres V0-V1 ($2-$3) (V0 ($2) est pour la partie haute et V1 ($3) pour la partie basse) :

Listing 1.373 : GCC 4.4.5 avec optimisation (listing assembleur)

```
        li      $3,-1867841536   # 0xffffffff90ab0000
        li      $2,305397760     # 0x12340000
        ori     $3,$3,0xcdef
        j       $31
        ori     $2,$2,0x5678
```

Listing 1.374 : GCC 4.4.5 avec optimisation (IDA)

```
       lui     $v1, 0x90AB
       lui     $v0, 0x1234
       li      $v1, 0x90ABCDEF
       jr      $ra
       li      $v0, 0x12345678
```

1.34.2 Passage d'arguments, addition, soustraction

```
#include <stdint.h>

uint64_t f_add (uint64_t a, uint64_t b)
{
        return a+b;
};

void f_add_test ()
{
#ifdef __GNUC__
        printf ("%lld\n", f_add(12345678901234, 23456789012345));
#else
        printf ("%I64d\n", f_add(12345678901234, 23456789012345));
#endif
};

uint64_t f_sub (uint64_t a, uint64_t b)
{
        return a-b;
};
```

x86

Listing 1.375 : MSVC 2012 /Ob1 avec optimisation

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f_add  PROC
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        adc     edx, DWORD PTR _b$[esp]
        ret     0
_f_add  ENDP

_f_add_test PROC
        push    5461            ; 00001555H
        push    1972608889      ; 75939f79H
        push    2874            ; 00000b3aH
        push    1942892530      ; 73ce2ff2H
        call    _f_add
        push    edx
        push    eax
        push    OFFSET $SG1436 ; '%I64d', 0aH, 00H
        call    _printf
        add     esp, 28
        ret     0
_f_add_test ENDP

_f_sub  PROC
        mov     eax, DWORD PTR _a$[esp-4]
        sub     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        sbb     edx, DWORD PTR _b$[esp]
        ret     0
_f_sub  ENDP
```

Nous voyons dans la fonction `f_add_test()` que chaque valeur 64-bit est passée en utilisant deux valeurs 32-bit, partie haute d'abord, puis partie basse.

L'addition et la soustraction se déroulent aussi par paire.

Pour l'addition, la partie basse 32-bit est d'abord additionnée. Si il y a eu une retenue pendant l'addition, le flag CF est mis.

L'instruction suivante ADC additionne les parties hautes, et ajoute aussi 1 si $CF = 1$.

La soustraction est aussi effectuée par paire. Le premier SUB peut aussi mettre le flag CF, qui doit être testé lors de l'instruction SBB suivante: Si le flag de retenue est mis, alors 1 est soustrait du résultat.

Il est facile de voir comment le résultat de la fonction `f_add()` est passé à `printf()`.

Listing 1.376 : GCC 4.8.1 -O1 -fno-inline

```
_f_add:
        mov     eax, DWORD PTR [esp+12]
```

```
        mov     edx, DWORD PTR [esp+16]
        add     eax, DWORD PTR [esp+4]
        adc     edx, DWORD PTR [esp+8]
        ret

_f_add_test:
        sub     esp, 28
        mov     DWORD PTR [esp+8], 1972608889   ; 75939f79H
        mov     DWORD PTR [esp+12], 5461        ; 00001555H
        mov     DWORD PTR [esp], 1942892530     ; 73ce2ff2H
        mov     DWORD PTR [esp+4], 2874         ; 00000b3aH
        call    _f_add
        mov     DWORD PTR [esp+4], eax
        mov     DWORD PTR [esp+8], edx
        mov     DWORD PTR [esp], OFFSET FLAT:LC0 ; "%lld\n"
        call    _printf
        add     esp, 28
        ret

_f_sub:
        mov     eax, DWORD PTR [esp+4]
        mov     edx, DWORD PTR [esp+8]
        sub     eax, DWORD PTR [esp+12]
        sbb     edx, DWORD PTR [esp+16]
        ret
```

Le code de GCC est le même.

ARM

Listing 1.377 : avec optimisation Keil 6/2013 (Mode ARM)

```
f_add PROC
        ADDS     r0,r0,r2
        ADC      r1,r1,r3
        BX       lr
        ENDP

f_sub PROC
        SUBS     r0,r0,r2
        SBC      r1,r1,r3
        BX       lr
        ENDP

f_add_test PROC
        PUSH     {r4,lr}
        LDR      r2,|L0.68| ; 0x75939f79
        LDR      r3,|L0.72| ; 0x00001555
        LDR      r0,|L0.76| ; 0x73ce2ff2
        LDR      r1,|L0.80| ; 0x00000b3a
        BL       f_add
        POP      {r4,lr}
        MOV      r2,r0
```

```
        MOV      r3,r1
        ADR      r0,|L0.84| ; "%I64d\n"
        B        __2printf
        ENDP

|L0.68|
        DCD      0x75939f79
|L0.72|
        DCD      0x00001555
|L0.76|
        DCD      0x73ce2ff2
|L0.80|
        DCD      0x00000b3a
|L0.84|
        DCB      "%I64d\n",0
```

La première valeur 64-bit est passée par la paire de registres R0 et R1, la seconde dans la paire de registres R2 et R3. ARM a aussi l'instruction ADC (qui compte le flag de retenue) et SBC («soustraction avec retenue »). Chose importante: lorsque les parties basses sont ajoutées/soustraites, les instructions ADDS et SUBS avec le suffixe -S sont utilisées. Le suffixe -S signifie «mettre les flags », et les flags (en particulier le flag de retenue) est ce dont les instructions suivantes ADC/SBC ont besoin. Autrement, les instructions sans le suffixe -S feraient le travail (ADD et SUB).

MIPS

Listing 1.378 : GCC 4.4.5 avec optimisation (IDA)

```
f_add:
; $a0 - partie haute de a
; $a1 - partie basse de a
; $a2 - partie haute de b
; $a3 - partie basse de b
                addu    $v1, $a3, $a1 ; ajouter les parties basses
                addu    $a0, $a2, $a0 ; ajouter les parties hautes
; est-ce qu'une retenue a été générée lors de l'addition des parties basses?
; si oui, mettre $v0 à 1
                sltu    $v0, $v1, $a3
                jr      $ra
; ajouter 1 à la partie haute du résultat si la retenue doit être générée:
                addu    $v0, $a0 ; slot de délai de branchement
; $v0 - partie haute du résultat
; $v1 - partie basse du résultat

f_sub:
; $a0 - partie haute de a
; $a1 - partie basse de a
; $a2 - partie haute de b
; $a3 - partie basse de b
                subu    $v1, $a1, $a3 ; soustraire les parties basses
                subu    $v0, $a0, $a2 ; soustraire les parties hautes
; est-ce qu'une retenue a été générée lors de la soustraction des parties
    basses?
```

```
; si oui, mettre $a0 à 1
                sltu    $a1, $v1
                jr      $ra
; soustraire 1 à la partie haute du résultat si la retenue doit être générée:
                subu    $v0, $a1 ; slot de délai de branchement
; $v0 - partie haute du résultat
; $v1 - partie basse du résultat

f_add_test:

var_10          = -0x10
var_4           = -4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+var_4($sp)
                sw      $gp, 0x20+var_10($sp)
                lui     $a1, 0x73CE
                lui     $a3, 0x7593
                li      $a0, 0xB3A
                li      $a3, 0x75939F79
                li      $a2, 0x1555
                jal     f_add
                li      $a1, 0x73CE2FF2
                lw      $gp, 0x20+var_10($sp)
                lui     $a0, ($LC0 >> 16)  # "%lld\n"
                lw      $t9, (printf & 0xFFFF)($gp)
                lw      $ra, 0x20+var_4($sp)
                la      $a0, ($LC0 & 0xFFFF)  # "%lld\n"
                move    $a3, $v1
                move    $a2, $v0
                jr      $t9
                addiu   $sp, 0x20

$LC0:           .ascii "%lld\n"<0>
```

MIPS n'a pas de registre de flags, donc il n'y a pas cette information après l'exécution des opérations arithmétiques. Donc il n'y a pas d'instructions comme ADC et SBB du x86. Pour savoir si le flag de retenue serait mis, une comparaison est faite (en utilisant l'instruction SLTU), qui met le registre de destination à 1 ou 0. Ce 1 ou ce 0 est ensuite ajouté ou soustrait au/du résultat final.

1.34.3 Multiplication, division

```
#include <stdint.h>

uint64_t f_mul (uint64_t a, uint64_t b)
{
        return a*b;
};
```

```
uint64_t f_div (uint64_t a, uint64_t b)
{
        return a/b;
};

uint64_t f_rem (uint64_t a, uint64_t b)
{
        return a % b;
};
```

x86

Listing 1.379 : MSVC 2013 /Ob1 avec optimisation

```
_a$ = 8 ; signe = 8
_b$ = 16 ; signe = 8
_f_mul  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __allmul ; multiplication long long
        pop     ebp
        ret     0
_f_mul  ENDP

_a$ = 8 ; signe = 8
_b$ = 16 ; signe = 8
_f_div  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __aulldiv ; division long long non signée
        pop     ebp
        ret     0
_f_div  ENDP

_a$ = 8 ; signe = 8
_b$ = 16 ; signe = 8
_f_rem  PROC
```

```
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __aullrem ; reste long long non signé
        pop     ebp
        ret     0
_f_rem  ENDP
```

La multiplication et la division sont des opérations plus complexes, donc en général le compilateur embarque des appels à des fonctions de bibliothèque les effectuant.

Ces fonctions sont décrites ici: .5 on page 1354.

Listing 1.380 : GCC 4.8.1 -fno-inline avec optimisation

```
_f_mul:
        push    ebx
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+16]
        mov     ebx, DWORD PTR [esp+12]
        mov     ecx, DWORD PTR [esp+20]
        imul    ebx, eax
        imul    ecx, edx
        mul     edx
        add     ecx, ebx
        add     edx, ecx
        pop     ebx
        ret

_f_div:
        sub     esp, 28
        mov     eax, DWORD PTR [esp+40]
        mov     edx, DWORD PTR [esp+44]
        mov     DWORD PTR [esp+8], eax
        mov     eax, DWORD PTR [esp+32]
        mov     DWORD PTR [esp+12], edx
        mov     edx, DWORD PTR [esp+36]
        mov     DWORD PTR [esp], eax
        mov     DWORD PTR [esp+4], edx
        call    ___udivdi3 ; division non signé
        add     esp, 28
        ret

_f_rem:
        sub     esp, 28
        mov     eax, DWORD PTR [esp+40]
        mov     edx, DWORD PTR [esp+44]
```

```
        mov     DWORD PTR [esp+8], eax
        mov     eax, DWORD PTR [esp+32]
        mov     DWORD PTR [esp+12], edx
        mov     edx, DWORD PTR [esp+36]
        mov     DWORD PTR [esp], eax
        mov     DWORD PTR [esp+4], edx
        call    ___umoddi3 ; modulo non signé
        add     esp, 28
        ret
```

GCC fait ce que l'on attend, mais le code multiplication est mis en ligne (inlined) directement dans la fonction, pensant que ça peut être plus efficace. GCC a des noms de fonctions de bibliothèque différents: .4 on page 1354.

ARM

Keil pour mode Thumb insère des appels à des sous-routines de bibliothèque:

Listing 1.381 : avec optimisation Keil 6/2013 (Mode Thumb)

```
||f_mul|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_lmul
        POP      {r4,pc}
        ENDP

||f_div|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_uldivmod
        POP      {r4,pc}
        ENDP

||f_rem|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_uldivmod
        MOVS     r0,r2
        MOVS     r1,r3
        POP      {r4,pc}
        ENDP
```

Keil pour mode ARM, d'un autre côté, est capable de produire le code de la multiplication 64-bit:

Listing 1.382 : avec optimisation Keil 6/2013 (Mode ARM)

```
||f_mul|| PROC
        PUSH     {r4,lr}
        UMULL    r12,r4,r0,r2
        MLA      r1,r2,r1,r4
        MLA      r1,r0,r3,r1
        MOV      r0,r12
        POP      {r4,pc}
        ENDP
```

```
||f_div|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_uldivmod
        POP      {r4,pc}
        ENDP

||f_rem|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_uldivmod
        MOV      r0,r2
        MOV      r1,r3
        POP      {r4,pc}
        ENDP
```

MIPS

GCC avec optimisation pour MIPS peut générer du code pour la multiplication 64-bit, mais doit appeler une routine de bibliothèque pour la division 64-bit:

Listing 1.383 : GCC 4.4.5 avec optimisation (IDA)

```
f_mul:
       mult    $a2, $a1
       mflo    $v0
       or      $at, $zero  ; NOP
       or      $at, $zero  ; NOP
       mult    $a0, $a3
       mflo    $a0
       addu    $v0, $a0
       or      $at, $zero  ; NOP
       multu   $a3, $a1
       mfhi    $a2
       mflo    $v1
       jr      $ra
       addu    $v0, $a2

f_div:

var_10 = -0x10
var_4  = -4

       lui     $gp, (__gnu_local_gp >> 16)
       addiu   $sp, -0x20
       la      $gp, (__gnu_local_gp & 0xFFFF)
       sw      $ra, 0x20+var_4($sp)
       sw      $gp, 0x20+var_10($sp)
       lw      $t9, (__udivdi3 & 0xFFFF)($gp)
       or      $at, $zero
       jalr    $t9
       or      $at, $zero
       lw      $ra, 0x20+var_4($sp)
       or      $at, $zero
```

```
        jr      $ra
        addiu   $sp, 0x20

f_rem:

var_10 = -0x10
var_4  = -4

        lui     $gp, (__gnu_local_gp >> 16)
        addiu   $sp, -0x20
        la      $gp, (__gnu_local_gp & 0xFFFF)
        sw      $ra, 0x20+var_4($sp)
        sw      $gp, 0x20+var_10($sp)
        lw      $t9, (__umoddi3 & 0xFFFF)($gp)
        or      $at, $zero
        jalr    $t9
        or      $at, $zero
        lw      $ra, 0x20+var_4($sp)
        or      $at, $zero
        jr      $ra
        addiu   $sp, 0x20
```

Il y a beaucoup de NOPs, sans doute des slots de délai de remplissage après l'instruction de multiplication (c'est plus lent que les autres instructions après tout).

1.34.4 Décalage à droite

```
#include <stdint.h>

uint64_t f (uint64_t a)
{
        return a>>7;
};
```

x86

Listing 1.384 : MSVC 2012 /Ob1 avec optimisation

```
_a$ = 8         ; size = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        shrd    eax, edx, 7
        shr     edx, 7
        ret     0
_f      ENDP
```

Listing 1.385 : GCC 4.8.1 -fno-inline avec optimisation

```
_f:
        mov     edx, DWORD PTR [esp+8]
```

```
        mov     eax, DWORD PTR [esp+4]
        shrd    eax, edx, 7
        shr     edx, 7
        ret
```

Le décalage se produit en deux passes: tout d'abord la partie basse est décalée, puis la partie haute. Mais la partie basse est décalée avec l'aide de l'instruction SHRD, elle décale la valeur de EAX de 7 bits, mais tire les nouveaux bits de EDX, i.e., de la partie haute. En d'autres mots, la valeur 64-bit dans la paire de registres EDX:EAX, dans son entier, est décalée de 7 bits et les 32 bits bas du résultat sont placés dans EAX. La partie haute est décalée en utilisant l'instruction plus populaire SHR : en effet, les bits libérés dans la partie haute doivent être remplis avec des zéros.

ARM

ARM n'a pas une instruction telle que SHRD en x86, donc le compilateur Keil fait cela en utilisant des simples décalages et des opérations OR :

Listing 1.386 : avec optimisation Keil 6/2013 (Mode ARM)

```
||f|| PROC
        LSR      r0,r0,#7
        ORR      r0,r0,r1,LSL #25
        LSR      r1,r1,#7
        BX       lr
        ENDP
```

Listing 1.387 : avec optimisation Keil 6/2013 (Mode Thumb)

```
||f|| PROC
        LSLS     r2,r1,#25
        LSRS     r0,r0,#7
        ORRS     r0,r0,r2
        LSRS     r1,r1,#7
        BX       lr
        ENDP
```

MIPS

GCC pour MIPS suit le même algorithme que Keil fait pour le mode Thumb:

Listing 1.388 : GCC 4.4.5 avec optimisation (IDA)

```
f:
       sll     $v0, $a0, 25
       srl     $v1, $a1, 7
       or      $v1, $v0, $v1
       jr      $ra
       srl     $v0, $a0, 7
```

1.34.5 Convertir une valeur 32-bit en 64-bit

```
#include <stdint.h>

int64_t f (int32_t a)
{
        return a;
};
```

x86

Listing 1.389 : MSVC 2012 avec optimisation

```
_a$ = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        cdq
        ret     0
_f      ENDP
```

Ici, nous nous heurtons à la nécessité d'étendre une valeur 32-bit signée en une 64-bit signée. Les valeurs non signées sont converties directement: tous les bits de la partie haute doivent être mis à 0. Mais ce n'est pas approprié pour les types de donnée signée: le signe doit être copié dans la partie haute du nombre résultant.

L'instruction CDQ fait cela ici, elle prend sa valeur d'entrée dans EAX, étend le signe sur 64-bit et laisse le résultat dans la paire de registres EDX :EAX. En d'autres mots, CDQ prend le signe du nombre dans EAX (en prenant le bit le plus significatif dans EAX), et suivant sa valeur, met tous les 32 bits de EDX à 0 ou 1. Cette opération est quelque peu similaire à l'instruction MOVSX.

ARM

Listing 1.390 : avec optimisation Keil 6/2013 (Mode ARM)

```
||f|| PROC
        ASR      r1,r0,#31
        BX       lr
        ENDP
```

Keil pour ARM est différent: il décale simplement arithmétiquement de 31 bits vers la droite la valeur en entrée. Comme nous le savons, le bit de signe est le MSB, et le décalage arithmétique copie le bit de signe dans les bits «apparus». Donc après «ASR r1,r0,#31», R1 contient 0xFFFFFFFF si la valeur d'entrée était négative et 0 sinon. R1 contient la partie haute de la valeur 64-bit résultante. En d'autres mots, ce code copie juste le MSB (bit de signe) de la valeur d'entrée dans R0 dans tous les bits de la partie haute 32-bit de la valeur 64-bit résultante.

MIPS

GCC pour MIPS fait la même chose que Keil a fait pour le mode ARM:

Listing 1.391 : GCC 4.4.5 avec optimisation (IDA)

```
f:
    sra     $v0, $a0, 31
    jr      $ra
    move    $v1, $a0
```

1.35 Cas d'une structure LARGE_INTEGER

Imaginez ceci: fin des années 1990, vous êtes Microsoft, et vous développez un nouvel OS *sérieux* (Windows NT), qui doit concurrencer les systèmes UNIX. Les plate-formes cibles sont à la fois les CPUs 32-bit et 64-bit. Et vous avez besoin d'un type de donnée entier 64-bit pour toutes sortes de besoins, à commencer par la structure FILETIME[168]

Le problème: tous les compilateurs C/C++ cibles ne supportent pas encore les entiers 64-bit (ceci se passe à la fin des années 1990). Sans aucun doute, ceci sera changé dans le futur (proche), mais pas maintenant. Que feriez-vous?

En lisant ceci, essayez d'arrêter (et/ou de fermer ce livre) et réfléchissez à comment vous résoudriez ce problème.

[168] https://docs.microsoft.com/en-us/windows/desktop/api/minwinbase/ns-minwinbase-filetime

Voici ce que fit Microsoft, quelque chose comme ceci[169] :

```
union ULARGE_INTEGER
{
    struct backward_compatibility
    {
        DWORD LowPart;
        DWORD HighPart;
    };
#ifdef NEW_FANCY_COMPILER_SUPPORTING_64_BIT
    ULONGLONG QuadPart;
#endif
};
```

Ceci est un fragment de 8 octets, qui peut être accédé par l'entier 64-bit QuadPart (si il est compilé avec un compilateur récent) ou en utilisant deux entiers 32-bit (si compilé avec un compilateur plus ancien).

Le champ QuadPart est simplement absent lorsque c'est compilé avec un vieux compilateur.

L'ordre est crucial: le premier champ (LowPart) correspond au 4 octets de la valeur 64-bit, le second (HighPart) au 4 octets hauts.

Microsoft a aussi ajouté des fonctions utilitaires pour les différentes opérations arithmétiques, de la même façon que je l'ai déjà décrit: 1.34 on page 509.

Et ceci provient du code source de Windows 2000 qui avait été divulgué:

Listing 1.392 : i386 arch

```
;++
;
; LARGE_INTEGER
; RtlLargeIntegerAdd (
; IN LARGE_INTEGER Addend1,
; IN LARGE_INTEGER Addend2
; )
;
; Routine Description:
;
; This function adds a signed large integer to a signed large integer and
; returns the signed large integer result.
;
; Arguments:
;
; (TOS+4) = Addend1 - first addend value
; (TOS+12) = Addend2 - second addend value
;
; Return Value:
;
; The large integer result is stored in (edx:eax)
;
;--
```

[169] Ce n'est pas un copier/coller du code source, j'ai écrit ceci

```
cPublicProc _RtlLargeIntegerAdd ,4
cPublicFpo 4,0

        mov     eax,[esp]+4             ; (eax)=add1.low
        add     eax,[esp]+12            ; (eax)=sum.low
        mov     edx,[esp]+8             ; (edx)=add1.hi
        adc     edx,[esp]+16            ; (edx)=sum.hi
        stdRET    _RtlLargeIntegerAdd

stdENDP _RtlLargeIntegerAdd
```

Listing 1.393 : MIPS arch

```
        LEAF_ENTRY(RtlLargeIntegerAdd)

        lw      t0,4 * 4(sp)            // get low part of addend2 value
        lw      t1,4 * 5(sp)            // get high part of addend2 value
        addu    t0,t0,a2                // add low parts of large integer
        addu    t1,t1,a3                // add high parts of large integer
        sltu    t2,t0,a2                // generate carry from low part
        addu    t1,t1,t2                // add carry to high part
        sw      t0,0(a0)                // store low part of result
        sw      t1,4(a0)                // store high part of result
        move    v0,a0                   // set function return register
        j       ra                      // return

        .end    RtlLargeIntegerAdd
```

Maintenant deux architectures 64-bit:

Listing 1.394 : Itanium arch

```
        LEAF_ENTRY(RtlLargeIntegerAdd)

        add         v0 = a0, a1                 // add both quadword ↙
    ↘ arguments
        LEAF_RETURN

        LEAF_EXIT(RtlLargeIntegerAdd)
```

Listing 1.395 : DEC Alpha arch

```
        LEAF_ENTRY(RtlLargeIntegerAdd)

        addq    a0, a1, v0              // add both quadword arguments
        ret     zero, (ra)              // return

        .end    RtlLargeIntegerAdd
```

Pas besoin d'utiliser des instructions 32-bit sur Itanium et DEC Alpha—qui soient déjà prêtes pour le 64-bit.

Et voici ce que l'on peut trouver dans Windows Research Kernel:

```
DECLSPEC_DEPRECATED_DDK         // Use native __int64 math
__inline
LARGE_INTEGER
NTAPI
RtlLargeIntegerAdd (
    LARGE_INTEGER Addend1,
    LARGE_INTEGER Addend2
    )
{
    LARGE_INTEGER Sum;

    Sum.QuadPart = Addend1.QuadPart + Addend2.QuadPart;
    return Sum;
}
```

Toutes ces fonctions pourront être supprimées (dans le futur), mais maintenant elles opèrent sur le champ QuadPart. Si ce morceau de code doit être compilé en utilisant un compilateur 32-bit moderne (qui supporte les entiers 64-bit), il générera deux additions 32-bit sous le capot. À partir de ce moment, les champs LowPart/HighPart pourront être supprimés de l'union/structure LARGE_INTEGER.

Utiliseriez-vous une telle technique aujourd'hui? Probablement pas, mais si quelqu'un avait besoin d'un type entier 128-bit, vous pourriez l'implémenter comme ceci.

Aussi, inutile de dire, ceci fonctionne grâce au *petit boutisme* (2.2 on page 586) (toutes les architectures pour lesquelles Windows NT a été développé sont *petit boutiste*. Cette astuce n'est pas possible sur une architecture *gros boutiste*.

1.36 SIMD

SIMD est un acronyme: *Single Instruction, Multiple Data* (simple instruction, multiple données).

Comme son nom le laisse entendre, cela traite des données multiples avec une seule instruction.

Comme le FPU, ce sous-système du CPU ressemble à un processeur séparé à l'intérieur du x86.

SIMD a commencé avec le MMX en x86. 8 nouveaux registres apparurent: MM0-MM7.

Chaque registre MMX contient 2 valeurs 32-bit, 4 valeurs 16-bit ou 8 octets. Par exemple, il est possible d'ajouter 8 valeurs 8-bit (octets) simultanément en ajoutant deux valeurs dans des registres MMX.

Un exemple simple est un éditeur graphique qui représente une image comme un tableau à deux dimensions. Lorsque l'utilisateur change la luminosité de l'image, l'éditeur doit ajouter ou soustraire un coefficient à/de chaque valeur du pixel. Dans un soucis de concision, si l'on dit que l'image est en niveau de gris et que chaque pixel est défini par un octet de 8-bit, alors il est possible de changer la luminosité de 8 pixels simultanément.

À propos, c'est la raison pour laquelle les instructions de *saturation* sont présentes en SIMD.

Lorsque l'utilisateur change la luminosité dans l'éditeur graphique, les dépassements au dessus ou en dessous ne sont pas souhaitables, donc il y a des instructions d'addition en SIMD qui n'additionnent pas si la valeur maximum est atteinte, etc.

Lorsque le MMX est apparu, ces registres étaient situés dans les registres du FPU. Il était seulement possible d'utiliser soit le FPU ou soit le MMX. On peut penser qu'Intel économisait des transistors, mais en fait, la raison d'une telle symbiose était plus simple —les anciens OSes qui n'étaient pas au courant de ces registres supplémentaires et ne les sauvaient pas lors du changement de contexte, mais sauvaient les registres FPU. Ainsi, CPU avec MMX + ancien OS + processus utilisant les capacités MMX fonctionnait toujours.

SSE—est une extension des registres SIMD à 128 bits, maintenant séparé du FPU.

AVX—une autre extension, à 256 bits.

Parlons maintenant de l'usage pratique.

Bien sûr, il s'agit de routines de copie en mémoire (`memcpy`), de comparaison de mémoire (`memcmp`) et ainsi de suite.

Un autre exemple: l'algorithme de chiffrement DES prend un bloc de 64-bit et une clef de 56-bit, chiffre le bloc et produit un résultat de 64-bit. L'algorithme DES peut être considéré comme un grand circuit électronique, avec des fils et des portes AND/OR/NOT.

Le bitslice DES[170] —est l'idée de traiter des groupes de blocs et de clés simultanément. Disons, une variable de type *unsigned int* en x86 peut contenir jusqu'à 32-bit, donc il est possible d'y stocker des résultats intermédiaires pour 32 paires de blocs-clé simultanément, en utilisant 64+56 variables de type *unsigned int*.

Il existe un utilitaire pour brute-forcer les mots de passe/hashes d'Oracle RDBMS (certains basés sur DES) en utilisant un algorithme bitslice DES légèrement modifié pour SSE2 et AVX—maintenant il est possible de chiffrer 128 ou 256 paires de blocs-clé simultanément.

http://conus.info/utils/ops_SIMD/

1.36.1 Vectorisation

La vectorisation[171], c'est lorsque, par exemple, vous avez une boucle qui prend une paire de tableaux en entrée et produit un tableau. Le corps de la boucle prend les valeurs dans les tableaux en entrée, fait quelque chose et met le résultat dans le tableau de sortie. La vectorisation est le fait de traiter plusieurs éléments simultanément.

La vectorisation n'est pas une nouvelle technologie: l'auteur de ce livre l'a vu au moins sur la série du super-calculateur Cray Y-MP de 1988 lorsqu'il jouait avec sa

[170] http://www.darkside.com.au/bitslice/
[171] Wikipédia: vectorisation

version «lite» le Cray Y-MP EL[172].

Par exemple:

```
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

Ce morceau de code prend des éléments de A et de B, les multiplie et sauve le résultat dans C.

Si chaque élément du tableau que nous avons est un *int* 32-bit, alors il est possible de charger 4 éléments de A dans un registre XMM 128-bit, 4 de B dans un autre registre XMM, et en exécutant *PMULLD* (*Multiply Packed Signed Dword Integers and Store Low Result*) et PMULHW (*Multiply Packed Signed Integers and Store High Result*), il est possible d'obtenir 4 produits 64-bit en une fois.

Ainsi, le nombre d'exécution du corps de la boucle est $1024/4$ au lieu de 1024, ce qui est 4 fois moins et, bien sûr, est plus rapide.

Exemple d'addition

Certains compilateurs peuvent effectuer la vectorisation automatiquement dans des cas simples, e.g., Intel C++[173].

Voici une fonction minuscule:

```
int f (int sz, int *ar1, int *ar2, int *ar3)
{
        for (int i=0; i<sz; i++)
                ar3[i]=ar1[i]+ar2[i];

        return 0;
};
```

Intel C++

Compilons la avec Intel C++ 11.1.051 win32:

```
icl intel.cpp /QaxSSE2 /Faintel.asm /Ox
```

Nous obtenons (dans IDA) :

```
; int __cdecl f(int, int *, int *, int *)
                public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near

var_10 = dword ptr -10h
sz     = dword ptr  4
```

[172] À distance. Il est installé dans le musée des super-calculateurs: http://www.cray-cyber.org
[173] Sur la vectorisation automatique d'Intel C++: Extrait: Vectorisation automatique efficace

```
ar1     = dword ptr  8
ar2     = dword ptr  0Ch
ar3     = dword ptr  10h

        push    edi
        push    esi
        push    ebx
        push    esi
        mov     edx, [esp+10h+sz]
        test    edx, edx
        jle     loc_15B
        mov     eax, [esp+10h+ar3]
        cmp     edx, 6
        jle     loc_143
        cmp     eax, [esp+10h+ar2]
        jbe     short loc_36
        mov     esi, [esp+10h+ar2]
        sub     esi, eax
        lea     ecx, ds:0[edx*4]
        neg     esi
        cmp     ecx, esi
        jbe     short loc_55

loc_36: ; CODE XREF: f(int,int *,int *,int *)+21
        cmp     eax, [esp+10h+ar2]
        jnb     loc_143
        mov     esi, [esp+10h+ar2]
        sub     esi, eax
        lea     ecx, ds:0[edx*4]
        cmp     esi, ecx
        jb      loc_143

loc_55: ; CODE XREF: f(int,int *,int *,int *)+34
        cmp     eax, [esp+10h+ar1]
        jbe     short loc_67
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        neg     esi
        cmp     ecx, esi
        jbe     short loc_7F

loc_67: ; CODE XREF: f(int,int *,int *,int *)+59
        cmp     eax, [esp+10h+ar1]
        jnb     loc_143
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        cmp     esi, ecx
        jb      loc_143

loc_7F: ; CODE XREF: f(int,int *,int *,int *)+65
        mov     edi, eax        ; edi = ar3
        and     edi, 0Fh        ; est-ce que ar3 est aligné sur 16-octets?
        jz      short loc_9A    ; oui
```

```
        test    edi, 3
        jnz     loc_162
        neg     edi
        add     edi, 10h
        shr     edi, 2

loc_9A: ; CODE XREF: f(int,int *,int *,int *)+84
        lea     ecx, [edi+4]
        cmp     edx, ecx
        jl      loc_162
        mov     ecx, edx
        sub     ecx, edi
        and     ecx, 3
        neg     ecx
        add     ecx, edx
        test    edi, edi
        jbe     short loc_D6
        mov     ebx, [esp+10h+ar2]
        mov     [esp+10h+var_10], ecx
        mov     ecx, [esp+10h+ar1]
        xor     esi, esi

loc_C1: ; CODE XREF: f(int,int *,int *,int *)+CD
        mov     edx, [ecx+esi*4]
        add     edx, [ebx+esi*4]
        mov     [eax+esi*4], edx
        inc     esi
        cmp     esi, edi
        jb      short loc_C1
        mov     ecx, [esp+10h+var_10]
        mov     edx, [esp+10h+sz]

loc_D6: ; CODE XREF: f(int,int *,int *,int *)+B2
        mov     esi, [esp+10h+ar2]
        lea     esi, [esi+edi*4] ; est-ce que ar2+i*4 est aligné sur
    16-octets?
        test    esi, 0Fh
        jz      short loc_109   ; oui!
        mov     ebx, [esp+10h+ar1]
        mov     esi, [esp+10h+ar2]

loc_ED: ; CODE XREF: f(int,int *,int *,int *)+105
        movdqu  xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
        movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 n'est pas aligné sur
    16-octet, donc le charger dans XMM0
        paddd   xmm1, xmm0
        movdqa  xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
        add     edi, 4
        cmp     edi, ecx
        jb      short loc_ED
        jmp     short loc_127

loc_109: ; CODE XREF: f(int,int *,int *,int *)+E3
        mov     ebx, [esp+10h+ar1]
```

```
        mov     esi, [esp+10h+ar2]

loc_111: ; CODE XREF: f(int,int *,int *,int *)+125
        movdqu  xmm0, xmmword ptr [ebx+edi*4]
        paddd   xmm0, xmmword ptr [esi+edi*4]
        movdqa  xmmword ptr [eax+edi*4], xmm0
        add     edi, 4
        cmp     edi, ecx
        jb      short loc_111

loc_127: ; CODE XREF: f(int,int *,int *,int *)+107
         ; f(int,int *,int *,int *)+164
        cmp     ecx, edx
        jnb     short loc_15B
        mov     esi, [esp+10h+ar1]
        mov     edi, [esp+10h+ar2]

loc_133: ; CODE XREF: f(int,int *,int *,int *)+13F
        mov     ebx, [esi+ecx*4]
        add     ebx, [edi+ecx*4]
        mov     [eax+ecx*4], ebx
        inc     ecx
        cmp     ecx, edx
        jb      short loc_133
        jmp     short loc_15B

loc_143: ; CODE XREF: f(int,int *,int *,int *)+17
         ; f(int,int *,int *,int *)+3A ...
        mov     esi, [esp+10h+ar1]
        mov     edi, [esp+10h+ar2]
        xor     ecx, ecx

loc_14D: ; CODE XREF: f(int,int *,int *,int *)+159
        mov     ebx, [esi+ecx*4]
        add     ebx, [edi+ecx*4]
        mov     [eax+ecx*4], ebx
        inc     ecx
        cmp     ecx, edx
        jb      short loc_14D

loc_15B: ; CODE XREF: f(int,int *,int *,int *)+A
         ; f(int,int *,int *,int *)+129 ...
        xor     eax, eax
        pop     ecx
        pop     ebx
        pop     esi
        pop     edi
        retn

loc_162: ; CODE XREF: f(int,int *,int *,int *)+8C
         ; f(int,int *,int *,int *)+9F
        xor     ecx, ecx
        jmp     short loc_127
```

```
?f@@YAHHPAH00@Z endp
```

Les instructions relatives à SSE2 sont:

- MOVDQU (*Move Unaligned Double Quadword* déplacer double quadruple mot non alignés)—charge juste 16 octets depuis la mémoire dans un registre XMM.
- PADDD (*Add Packed Integers* ajouter entier packé)—ajoute 4 paires de nombres 32-bit et laisse le résultat dans le premier opérande. À propos, aucune exception n'est levée en cas de débordement et aucun flag n'est mis, seuls les 32-bit bas du résultat sont stockés. Si un des opérandes de PADDD est l'adresse d'une valeur en mémoire, alors l'adresse doit être alignée sur une limite de 16 octets. Si elle n'est pas alignée, une exception est levée.
- MOVDQA (*Move Aligned Double Quadword*) est la même chose que MOVDQU, mais nécessite que l'adresse de la valeur en mémoire soit alignée sur une limite de 16 octets. Si elle n'est pas alignée, une exception est levée. MOVDQA fonctionne plus vite que MOVDQU, mais nécessite la condition qui vient d'être écrite.

Donc, ces instructions SSE2 sont exécutées seulement dans le cas où il y a plus de 4 paires à traiter et que le pointeur ar3 est aligné sur une limite de 16 octets.

Ainsi, si ar2 est également aligné sur une limite de 16 octets, ce morceau de code sera exécuté:

```
movdqu  xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
paddd   xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
movdqa  xmmword ptr [eax+edi*4], xmm0 ; ar3+i*4
```

Autrement, la valeur de ar2 est chargée dans XMM0 avec MOVDQU, qui ne nécessite pas que le pointeur soit aligné, mais peut s'exécuter plus lentement.

```
movdqu  xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 n'est pas aligné sur
    16-octet, donc le charger dans XMM0
paddd   xmm1, xmm0
movdqa  xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
```

Dans tous les autres cas, le code non-SSE2 sera exécuté.

GCC

GCC peut aussi vectoriser dans des cas simples[174], si l'option -O3 est utilisée et le support de SSE2 activé: -msse2.

Ce que nous obtenons (GCC 4.4.1) :

```
; f(int, int *, int *, int *)
            public _Z1fiPiS_S_
_Z1fiPiS_S_ proc near
```

[174] Plus sur le support de la vectorisation dans GCC: http://gcc.gnu.org/projects/tree-ssa/vectorization.html

```
var_18      = dword ptr -18h
var_14      = dword ptr -14h
var_10      = dword ptr -10h
arg_0       = dword ptr  8
arg_4       = dword ptr  0Ch
arg_8       = dword ptr  10h
arg_C       = dword ptr  14h

            push    ebp
            mov     ebp, esp
            push    edi
            push    esi
            push    ebx
            sub     esp, 0Ch
            mov     ecx, [ebp+arg_0]
            mov     esi, [ebp+arg_4]
            mov     edi, [ebp+arg_8]
            mov     ebx, [ebp+arg_C]
            test    ecx, ecx
            jle     short loc_80484D8
            cmp     ecx, 6
            lea     eax, [ebx+10h]
            ja      short loc_80484E8

loc_80484C1: ; CODE XREF: f(int,int *,int *,int *)+4B
             ; f(int,int *,int *,int *)+61 ...
            xor     eax, eax
            nop
            lea     esi, [esi+0]

loc_80484C8: ; CODE XREF: f(int,int *,int *,int *)+36
            mov     edx, [edi+eax*4]
            add     edx, [esi+eax*4]
            mov     [ebx+eax*4], edx
            add     eax, 1
            cmp     eax, ecx
            jnz     short loc_80484C8

loc_80484D8: ; CODE XREF: f(int,int *,int *,int *)+17
             ; f(int,int *,int *,int *)+A5
            add     esp, 0Ch
            xor     eax, eax
            pop     ebx
            pop     esi
            pop     edi
            pop     ebp
            retn

            align 8

loc_80484E8: ; CODE XREF: f(int,int *,int *,int *)+1F
            test    bl, 0Fh
```

```
            jnz     short loc_80484C1
            lea     edx, [esi+10h]
            cmp     ebx, edx
            jbe     loc_8048578

loc_80484F8: ; CODE XREF: f(int,int *,int *,int *)+E0
            lea     edx, [edi+10h]
            cmp     ebx, edx
            ja      short loc_8048503
            cmp     edi, eax
            jbe     short loc_80484C1

loc_8048503: ; CODE XREF: f(int,int *,int *,int *)+5D
            mov     eax, ecx
            shr     eax, 2
            mov     [ebp+var_14], eax
            shl     eax, 2
            test    eax, eax
            mov     [ebp+var_10], eax
            jz      short loc_8048547
            mov     [ebp+var_18], ecx
            mov     ecx, [ebp+var_14]
            xor     eax, eax
            xor     edx, edx
            nop

loc_8048520: ; CODE XREF: f(int,int *,int *,int *)+9B
            movdqu  xmm1, xmmword ptr [edi+eax]
            movdqu  xmm0, xmmword ptr [esi+eax]
            add     edx, 1
            paddd   xmm0, xmm1
            movdqa  xmmword ptr [ebx+eax], xmm0
            add     eax, 10h
            cmp     edx, ecx
            jb      short loc_8048520
            mov     ecx, [ebp+var_18]
            mov     eax, [ebp+var_10]
            cmp     ecx, eax
            jz      short loc_80484D8

loc_8048547: ; CODE XREF: f(int,int *,int *,int *)+73
            lea     edx, ds:0[eax*4]
            add     esi, edx
            add     edi, edx
            add     ebx, edx
            lea     esi, [esi+0]

loc_8048558: ; CODE XREF: f(int,int *,int *,int *)+CC
            mov     edx, [edi]
            add     eax, 1
            add     edi, 4
            add     edx, [esi]
            add     esi, 4
```

```
            mov     [ebx], edx
            add     ebx, 4
            cmp     ecx, eax
            jg      short loc_8048558
            add     esp, 0Ch
            xor     eax, eax
            pop     ebx
            pop     esi
            pop     edi
            pop     ebp
            retn

loc_8048578: ; CODE XREF: f(int,int *,int *,int *)+52
            cmp     eax, esi
            jnb     loc_80484C1
            jmp     loc_80484F8
_Z1fiPiS_S_ endp
```

Presque le même, toutefois, pas aussi méticuleux qu'Intel C++.

Exemple de copie de mémoire

Revoyons le simple exemple memcpy() (1.22.2 on page 256) :

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
        size_t i;
        for (i=0; i<cnt; i++)
                dst[i]=src[i];
};
```

Et ce que les optimisations de GCC 4.9.1 font:

Listing 1.396 : GCC 4.9.1 x64 avec optimisation

```
my_memcpy:
; RDI = adresse de destination
; RSI = adresse source
; RDX = taille du bloc
        test    rdx, rdx
        je      .L41
        lea     rax, [rdi+16]
        cmp     rsi, rax
        lea     rax, [rsi+16]
        setae   cl
        cmp     rdi, rax
        setae   al
        or      cl, al
        je      .L13
        cmp     rdx, 22
        jbe     .L13
```

```
        mov     rcx, rsi
        push    rbp
        push    rbx
        neg     rcx
        and     ecx, 15
        cmp     rcx, rdx
        cmova   rcx, rdx
        xor     eax, eax
        test    rcx, rcx
        je      .L4
        movzx   eax, BYTE PTR [rsi]
        cmp     rcx, 1
        mov     BYTE PTR [rdi], al
        je      .L15
        movzx   eax, BYTE PTR [rsi+1]
        cmp     rcx, 2
        mov     BYTE PTR [rdi+1], al
        je      .L16
        movzx   eax, BYTE PTR [rsi+2]
        cmp     rcx, 3
        mov     BYTE PTR [rdi+2], al
        je      .L17
        movzx   eax, BYTE PTR [rsi+3]
        cmp     rcx, 4
        mov     BYTE PTR [rdi+3], al
        je      .L18
        movzx   eax, BYTE PTR [rsi+4]
        cmp     rcx, 5
        mov     BYTE PTR [rdi+4], al
        je      .L19
        movzx   eax, BYTE PTR [rsi+5]
        cmp     rcx, 6
        mov     BYTE PTR [rdi+5], al
        je      .L20
        movzx   eax, BYTE PTR [rsi+6]
        cmp     rcx, 7
        mov     BYTE PTR [rdi+6], al
        je      .L21
        movzx   eax, BYTE PTR [rsi+7]
        cmp     rcx, 8
        mov     BYTE PTR [rdi+7], al
        je      .L22
        movzx   eax, BYTE PTR [rsi+8]
        cmp     rcx, 9
        mov     BYTE PTR [rdi+8], al
        je      .L23
        movzx   eax, BYTE PTR [rsi+9]
        cmp     rcx, 10
        mov     BYTE PTR [rdi+9], al
        je      .L24
        movzx   eax, BYTE PTR [rsi+10]
        cmp     rcx, 11
        mov     BYTE PTR [rdi+10], al
```

```
        je      .L25
        movzx   eax, BYTE PTR [rsi+11]
        cmp     rcx, 12
        mov     BYTE PTR [rdi+11], al
        je      .L26
        movzx   eax, BYTE PTR [rsi+12]
        cmp     rcx, 13
        mov     BYTE PTR [rdi+12], al
        je      .L27
        movzx   eax, BYTE PTR [rsi+13]
        cmp     rcx, 15
        mov     BYTE PTR [rdi+13], al
        jne     .L28
        movzx   eax, BYTE PTR [rsi+14]
        mov     BYTE PTR [rdi+14], al
        mov     eax, 15
.L4:
        mov     r10, rdx
        lea     r9, [rdx-1]
        sub     r10, rcx
        lea     r8, [r10-16]
        sub     r9, rcx
        shr     r8, 4
        add     r8, 1
        mov     r11, r8
        sal     r11, 4
        cmp     r9, 14
        jbe     .L6
        lea     rbp, [rsi+rcx]
        xor     r9d, r9d
        add     rcx, rdi
        xor     ebx, ebx
.L7:
        movdqa  xmm0, XMMWORD PTR [rbp+0+r9]
        add     rbx, 1
        movups  XMMWORD PTR [rcx+r9], xmm0
        add     r9, 16
        cmp     rbx, r8
        jb      .L7
        add     rax, r11
        cmp     r10, r11
        je      .L1
.L6:
        movzx   ecx, BYTE PTR [rsi+rax]
        mov     BYTE PTR [rdi+rax], cl
        lea     rcx, [rax+1]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+1+rax]
        mov     BYTE PTR [rdi+1+rax], cl
        lea     rcx, [rax+2]
        cmp     rdx, rcx
        jbe     .L1
```

```
        movzx   ecx, BYTE PTR [rsi+2+rax]
        mov     BYTE PTR [rdi+2+rax], cl
        lea     rcx, [rax+3]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+3+rax]
        mov     BYTE PTR [rdi+3+rax], cl
        lea     rcx, [rax+4]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+4+rax]
        mov     BYTE PTR [rdi+4+rax], cl
        lea     rcx, [rax+5]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+5+rax]
        mov     BYTE PTR [rdi+5+rax], cl
        lea     rcx, [rax+6]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+6+rax]
        mov     BYTE PTR [rdi+6+rax], cl
        lea     rcx, [rax+7]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+7+rax]
        mov     BYTE PTR [rdi+7+rax], cl
        lea     rcx, [rax+8]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+8+rax]
        mov     BYTE PTR [rdi+8+rax], cl
        lea     rcx, [rax+9]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+9+rax]
        mov     BYTE PTR [rdi+9+rax], cl
        lea     rcx, [rax+10]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+10+rax]
        mov     BYTE PTR [rdi+10+rax], cl
        lea     rcx, [rax+11]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+11+rax]
        mov     BYTE PTR [rdi+11+rax], cl
        lea     rcx, [rax+12]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+12+rax]
        mov     BYTE PTR [rdi+12+rax], cl
        lea     rcx, [rax+13]
```

```
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+13+rax]
        mov     BYTE PTR [rdi+13+rax], cl
        lea     rcx, [rax+14]
        cmp     rdx, rcx
        jbe     .L1
        movzx   edx, BYTE PTR [rsi+14+rax]
        mov     BYTE PTR [rdi+14+rax], dl
.L1:
        pop     rbx
        pop     rbp
.L41:
        rep ret
.L13:
        xor     eax, eax
.L3:
        movzx   ecx, BYTE PTR [rsi+rax]
        mov     BYTE PTR [rdi+rax], cl
        add     rax, 1
        cmp     rax, rdx
        jne     .L3
        rep ret
.L28:
        mov     eax, 14
        jmp     .L4
.L15:
        mov     eax, 1
        jmp     .L4
.L16:
        mov     eax, 2
        jmp     .L4
.L17:
        mov     eax, 3
        jmp     .L4
.L18:
        mov     eax, 4
        jmp     .L4
.L19:
        mov     eax, 5
        jmp     .L4
.L20:
        mov     eax, 6
        jmp     .L4
.L21:
        mov     eax, 7
        jmp     .L4
.L22:
        mov     eax, 8
        jmp     .L4
.L23:
        mov     eax, 9
        jmp     .L4
```

```
.L24:
        mov     eax, 10
        jmp     .L4
.L25:
        mov     eax, 11
        jmp     .L4
.L26:
        mov     eax, 12
        jmp     .L4
.L27:
        mov     eax, 13
        jmp     .L4
```

1.36.2 Implémentation SIMD de `strlen()`

Il faut noter que les instructions SIMD peuvent être insérées en code C/C++ via des macros[175] spécifiques. Pour MSVC, certaines d'entre elles se trouvent dans le fichier `intrin.h`.

Il est possible d'implémenter la fonction `strlen()`[176] en utilisant des instructions SIMD qui fonctionne 2-2.5 fois plus vite que l'implémentation habituelle. Cette fonction charge 16 caractères dans un registre XMM et teste chacun d'eux avec zéro. [177].

```
size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s=str;
    bool str_is_aligned=(((unsigned int)str)&0xFFFFFFF0) == (unsigned int)⤦
    ↳ str;

    if (str_is_aligned==false)
        return strlen (str);

    __m128i xmm0 = _mm_setzero_si128();
    __m128i xmm1;
    int mask = 0;

    for (;;)
    {
        xmm1 = _mm_load_si128((__m128i *)s);
        xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
        if ((mask = _mm_movemask_epi8(xmm1)) != 0)
        {
            unsigned long pos;
            _BitScanForward(&pos, mask);
            len += (size_t)pos;

            break;
```

[175] MSDN: particularités MMX, SSE, et SSE2
[176] strlen() —fonction de la bibliothèque C standard pour calculer la longueur d'une chaîne
[177] L'exemple est basé sur le code source de: http://www.strchr.com/sse2_optimised_strlen.

```
        }
        s += sizeof(__m128i);
        len += sizeof(__m128i);
    };

    return len;
}
```

Compilons la avec MSVC 2010 avec l'option /Ox :

Listing 1.397 : MSVC 2010 avec optimisation

```
_pos$75552 = -4          ; taille = 4
_str$ = 8                ; taille = 4
?strlen_sse2@@YAIPBD@Z PROC ; strlen_sse2

    push     ebp
    mov      ebp, esp
    and      esp, -16            ; fffffff0H
    mov      eax, DWORD PTR _str$[ebp]
    sub      esp, 12             ; 0000000cH
    push     esi
    mov      esi, eax
    and      esi, -16            ; fffffff0H
    xor      edx, edx
    mov      ecx, eax
    cmp      esi, eax
    je       SHORT $LN4@strlen_sse
    lea      edx, DWORD PTR [eax+1]
    npad     3 ; aligner le prochain label
$LL11@strlen_sse:
    mov      cl, BYTE PTR [eax]
    inc      eax
    test     cl, cl
    jne      SHORT $LL11@strlen_sse
    sub      eax, edx
    pop      esi
    mov      esp, ebp
    pop      ebp
    ret      0
$LN4@strlen_sse:
    movdqa   xmm1, XMMWORD PTR [eax]
    pxor     xmm0, xmm0
    pcmpeqb  xmm1, xmm0
    pmovmskb eax, xmm1
    test     eax, eax
    jne      SHORT $LN9@strlen_sse
$LL3@strlen_sse:
    movdqa   xmm1, XMMWORD PTR [ecx+16]
    add      ecx, 16                    ; 00000010H
    pcmpeqb  xmm1, xmm0
    add      edx, 16                    ; 00000010H
    pmovmskb eax, xmm1
    test     eax, eax
```

```
    je       SHORT $LL3@strlen_sse
$LN9@strlen_sse:
    bsf      eax, eax
    mov      ecx, eax
    mov      DWORD PTR _pos$75552[esp+16], eax
    lea      eax, DWORD PTR [ecx+edx]
    pop      esi
    mov      esp, ebp
    pop      ebp
    ret      0
?strlen_sse2@@YAIPBD@Z ENDP                      ; strlen_sse2
```

Comment est-ce que ça fonctionne? Premièrement, nous devons comprendre la but de la fonction. Elle calcule la longueur de la chaîne C, mais nous pouvons utiliser différents termes: sa tâche est de chercher l'octet zéro, et de calculer sa position relativement au début de la chaîne.

Premièrement, nous testons si le pointeur `str` est aligné sur une limite de 16 octets. Si non, nous appelons l'implémentation générique de `strlen()`.

Puis, nous chargeons les 16 octets suivants dans le registre `XMM1` en utilisant `MOVDQA`.

Un lecteur observateur pourrait demander, pourquoi `MOVDQU` ne pourrait pas être utilisé ici, puisqu'il peut charger des données depuis la mémoire quelque soit l'alignement du pointeur?

Oui, cela pourrait être fait comme ça: si le pointeur est aligné, charger les données en utilisant `MOVDQA`, si non —utiliser `MOVDQU` moins rapide.

Mais ici nous pourrions rencontrer une autre restriction:

Dans la série d'OS Windows NT (mais pas seulement), la mémoire est allouée par pages de 4 KiB (4096 octets). Chaque processus win32 a 4GiB de disponible, mais en fait, seulement une partie de l'espace d'adressage est connecté à de la mémoire réelle. Si le processus accède a un bloc mémoire absent, une exception est levée. C'est comme cela que fonctionnent les VM[178].

Donc, une fonction qui charge 16 octets à la fois peut dépasser la limite d'un bloc de mémoire allouée. Disons que l'OS a alloué 8192 (0x2000) octets à l'adresse 0x008c0000. Ainsi, le bloc comprend les octets démarrant à l'adresse 0x008c0000 jusqu'à 0x008c1fff inclus.

Après ce bloc, c'est à dire à partir de l'adresse 0x008c2000 il n'y a rien du tout, e.g. l'OS n'y a pas alloué de mémoire. Toutes tentatives d'accéder à la mémoire à partir de cette adresse va déclencher une exception.

Et maintenant, considérons l'exemple dans lequel le programme possède une chaîne contenant 5 caractères presque à la fin d'un bloc, et ce n'est pas un crime.

[178]Wikipédia

0x008c1ff8	'h'
0x008c1ff9	'e'
0x008c1ffa	'l'
0x008c1ffb	'l'
0x008c1ffc	'o'
0x008c1ffd	'\x00'
0x008c1ffe	random noise
0x008c1fff	random noise

Donc, dans des conditions normales, le programme appelle `strlen()`, en lui passant un pointeur sur la chaîne `'hello'` se trouvant en mémoire à l'adresse 0x008c1ff8. `strlen()` lit un octet à la fois jusqu'à 0x008c1ffd, où se trouve un octet à zéro, et puis s'arrête.

Maintenant, si nous implémentons notre propre `strlen()` lisant 16 octets à la fois, à partir de n'importe quelle adresse, alignée ou pas, `MOVDQU` pourrait essayer de charger 16 octets à la fois depuis l'adresse 0x008c1ff8 jusqu'à 0x008c2008, et ainsi déclencherait une exception. Cette situation peut être évitée, bien sûr.

Nous allons donc ne travailler qu'avec des adresses alignées sur une limite de 16 octets, ce qui en combinaison avec la connaissance que les pages de l'OS sont en général alignées sur une limite de 16 octets nous donne quelques garanties que notre fonction ne va pas lire de la mémoire non allouée.

Retournons à notre fonction.

`_mm_setzero_si128()`—est une macro générant `pxor xmm0, xmm0` —elle efface juste le registre `XMM0`.

`_mm_load_si128()`—est une macro pour `MOVDQA`, elle charge 16 octets depuis l'adresse dans le registre XMM1.

`_mm_cmpeq_epi8()`—est une macro pour PCMPEQB, une instruction qui compare deux registres XMM par octet.

Et si l'un des octets est égal à celui dans l'autre registre, il y aura `0xff` à ce point dans le résultat ou 0 sinon.

Par exemple:

```
XMM1: 0x11223344556677880000000000000000
XMM0: 0x11ab3444007877881111111111111111
```

Après l'exécution de `pcmpeqb xmm1, xmm0`, le registre XMM1 contient:

```
XMM1: 0xff0000ff0000ffff0000000000000000
```

Dans notre cas, cette instruction compare chacun des 16 octets avec un bloc de 16 octets à zéro, qui ont été mis dans le registre `XMM0` par `pxor xmm0, xmm0`.

La macro suivante est `_mm_movemask_epi8()` —qui est l'instruction PMOVMSKB.

Elle est très utile avec PCMPEQB.

```
pmovmskb eax, xmm1
```

Cette instruction met d'abord le premier bit d'EAX à 1 si le bit le plus significatif du premier octet dans `XMM1` est 1. En d'autres mots, si le premier octet du registre `XMM1` est `0xff`, alors le premier bit de EAX sera 1 aussi.

Si le second octet du registre `XMM1` est `0xff`, alors le second bit de EAX sera mis à 1 aussi. En d'autres mots, cette instruction répond à la question «quels octets de `XMM1` ont le bit le plus significatif à 1 ou sont plus grand que `0x7f`?» et renvoie 16 bits dans le registre EAX. Les autres bits du registre EAX sont mis à zéro.

À propos, ne pas oublier cette bizarrerie dans notre algorithme. Il pourrait y avoir 16 octets dans l'entrée, comme:

15	14	13	12	11	10	9 … 3	2	1 … 0
'h'	'e'	'l'	'l'	'o'	0	déchets	0	déchets

Il s'agit de la chaîne `'hello'`, terminée par un zéro, et du bruit aléatoire dans la mémoire.

Si nous chargeons ces 16 octets dans `XMM1` et les comparons avec ceux à zéro dans `XMM0`, nous obtenons quelque chose comme [179] :

`XMM1: 0x0000ff00000000000000ff0000000000`

Cela signifie que cette instruction a trouvé deux octets à zéro, et ce n'est pas surprenant.

`PMOVMSKB` dans notre cas va mettre EAX à
0b0010000000100000.

Bien sûr, notre fonction doit seulement prendre le premier octet à zéro et ignorer le reste.

L'instruction suivante est BSF (*Bit Scan Forward*).

Cette instruction trouve le premier bit mis à 1 et met sa position dans le premier opérande.

`EAX=0b0010000000100000`

Après l'exécution de `bsf eax, eax`, EAX contient 5, signifiant que 1 a été trouvé au bit d'index 5 (en commençant à zéro).

MSVC a une macro pour cette instruction: `_BitScanForward`.

Maintenant c'est simple. Si un octet à zéro a été trouvé, sa position est ajoutée à ce que nous avions déjà compté et nous pouvons renvoyer le résultat.

Presque tout.

À propos, il faut aussi noter que le compilateur MSVC a généré deux corps de boucle côte-à-côte, afin d'optimiser.

Et aussi, SSE 4.2 (apparu dans les Intel Core i7) offre encore plus d'instructions avec lesquelles ces manipulations de chaîne sont encore plus facile: `http://www.strchr.com/strcmp_and_strlen_using_sse_4.2`

[179] Un ordre de MSB à LSB[180] est utilisé ici.

1.37 64 bits

1.37.1 x86-64

Il s'agit d'une extension à 64 bits de l'architecture x86.

Pour l'ingénierie inverse, les changements les plus significatifs sont:

- La plupart des registres (à l'exception des registres FPU et SIMD) ont été étendus à 64 bits et leur nom préfixé de la lettre R. 8 registres ont également été ajoutés. Les GPR sont donc désormais: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15.

  Les *anciens* registres restent accessibles de la manière habituelle. Ainsi, l'utilisation de EAX donne accès aux 32 bits de poids faible du registre RAX :

Octet d'indice							
7	6	5	4	3	2	1	0
RAX^{x64}							
				EAX			
						AX	
						AH	AL

  Les nouveaux registres R8-R15 possèdent eux aussi des sous-parties : R8D-R15D (pour les 32 bits de poids faible), R8W-R15W (16 bits de poids faible), R8L-R15L (8 bits de poids faible).

Octet d'indice							
7	6	5	4	3	2	1	0
R8							
				R8D			
						R8W	
							R8L

  Les registres SIMD ont vu leur nombre passé de 8 à 16: XMM0-XMM15.

- En environnement Win64, la convention d'appel de fonctions est légèrement différente et ressemble à la convention fastcall (6.1.3 on page 955). Les 4 premiers arguments sont stockés dans les registres RCX, RDX, R8 et R9. Les arguments suivants —sur la pile. La fonction appelante doit également allouer 32 octets pour utilisation par la fonction appelée qui pourra y sauvegarder les registres contenant les 4 premiers arguments. Les fonctions les plus simples peuvent utiliser les arguments directement depuis les registres. En revanche, les fonctions plus complexes peuvent sauvegarder ces registres sur la pile.

  L'ABI System V AMD64 (Linux, *BSD, Mac OS X)[Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [181]ressemble elle aussi à la convention fastcall. Elle utilise 6 registres RDI, RSI, RDX, RCX, R8, R9 pour les 6 premiers arguments. Tous les suivants sont passés sur la pile.

[181]Aussi disponible en https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

Référez-vous également à la section sur les conventions d'appel (6.1 on page 953).

- Pour des raisons de compatibilité, le type C/C++ *int* conserve sa taille de 32bits.
- Tous les pointeurs sont désormais sur 64 bits.

Dans la mesure où le nombre de registres a doublé, les compilateurs disposent de plus de marge de manœuvre en matière d'allocation des registres. Pour nous, il en résulte que le code généré contient moins de variables locales.

Par exemple, la fonction qui calcule la première S-box de l'algorithme de chiffrage DES traite en une fois au moyen de la méthode DES bitslice des valeurs de 32/64/128/256 bits (en fonction du type DES_type (uint32, uint64, SSE2 or AVX)). Pour en savoir plus sur cette technique, voyez (1.36 on page 526) :

```
/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for any purpose,
 * so long as its origin is acknowledged.
 *
 * Produced by Matthew Kwan - March 1998
 */

#ifdef _WIN64
#define DES_type unsigned __int64
#else
#define DES_type unsigned int
#endif

void
s1 (
    DES_type    a1,
    DES_type    a2,
    DES_type    a3,
    DES_type    a4,
    DES_type    a5,
    DES_type    a6,
    DES_type    *out1,
    DES_type    *out2,
    DES_type    *out3,
    DES_type    *out4
) {
    DES_type    x1, x2, x3, x4, x5, x6, x7, x8;
    DES_type    x9, x10, x11, x12, x13, x14, x15, x16;
    DES_type    x17, x18, x19, x20, x21, x22, x23, x24;
    DES_type    x25, x26, x27, x28, x29, x30, x31, x32;
    DES_type    x33, x34, x35, x36, x37, x38, x39, x40;
    DES_type    x41, x42, x43, x44, x45, x46, x47, x48;
    DES_type    x49, x50, x51, x52, x53, x54, x55, x56;

    x1 = a3 & ~a5;
    x2 = x1 ^ a4;
    x3 = a3 & ~a4;
    x4 = x3 | a5;
```

```
    x5 = a6 & x4;
    x6 = x2 ^ x5;
    x7 = a4 & ~a5;
    x8 = a3 ^ a4;
    x9 = a6 & ~x8;
    x10 = x7 ^ x9;
    x11 = a2 | x10;
    x12 = x6 ^ x11;
    x13 = a5 ^ x5;
    x14 = x13 & x8;
    x15 = a5 & ~a4;
    x16 = x3 ^ x14;
    x17 = a6 | x16;
    x18 = x15 ^ x17;
    x19 = a2 | x18;
    x20 = x14 ^ x19;
    x21 = a1 & x20;
    x22 = x12 ^ ~x21;
    *out2 ^= x22;
    x23 = x1 | x5;
    x24 = x23 ^ x8;
    x25 = x18 & ~x2;
    x26 = a2 & ~x25;
    x27 = x24 ^ x26;
    x28 = x6 | x7;
    x29 = x28 ^ x25;
    x30 = x9 ^ x24;
    x31 = x18 & ~x30;
    x32 = a2 & x31;
    x33 = x29 ^ x32;
    x34 = a1 & x33;
    x35 = x27 ^ x34;
    *out4 ^= x35;
    x36 = a3 & x28;
    x37 = x18 & ~x36;
    x38 = a2 | x3;
    x39 = x37 ^ x38;
    x40 = a3 | x31;
    x41 = x24 & ~x37;
    x42 = x41 | x3;
    x43 = x42 & ~a2;
    x44 = x40 ^ x43;
    x45 = a1 & ~x44;
    x46 = x39 ^ ~x45;
    *out1 ^= x46;
    x47 = x33 & ~x9;
    x48 = x47 ^ x39;
    x49 = x4 ^ x36;
    x50 = x49 & ~x5;
    x51 = x42 | x18;
    x52 = x51 ^ a5;
    x53 = a2 & ~x52;
    x54 = x50 ^ x53;
```

```
    x55 = a1 | x54;
    x56 = x48 ^ ~x55;
    *out3 ^= x56;
}
```

Cette fonction contient de nombreuses variables locales, mais toutes ne se retrouveront pas dans la pile. Compilons ce fichier avec MSVC 2008 et l'option /Ox :

Listing 1.398 : MSVC 2008 avec optimisation

```
PUBLIC    _s1
; Function compile flags: /Ogtpy
_TEXT    SEGMENT
_x6$ = -20          ; size = 4
_x3$ = -16          ; size = 4
_x1$ = -12          ; size = 4
_x8$ = -8           ; size = 4
_x4$ = -4           ; size = 4
_a1$ = 8            ; size = 4
_a2$ = 12           ; size = 4
_a3$ = 16           ; size = 4
_x33$ = 20          ; size = 4
_x7$ = 20           ; size = 4
_a4$ = 20           ; size = 4
_a5$ = 24           ; size = 4
tv326 = 28          ; size = 4
_x36$ = 28          ; size = 4
_x28$ = 28          ; size = 4
_a6$ = 28           ; size = 4
_out1$ = 32         ; size = 4
_x24$ = 36          ; size = 4
_out2$ = 36         ; size = 4
_out3$ = 40         ; size = 4
_out4$ = 44         ; size = 4
_s1    PROC
    sub    esp, 20                  ; 00000014H
    mov    edx, DWORD PTR _a5$[esp+16]
    push   ebx
    mov    ebx, DWORD PTR _a4$[esp+20]
    push   ebp
    push   esi
    mov    esi, DWORD PTR _a3$[esp+28]
    push   edi
    mov    edi, ebx
    not    edi
    mov    ebp, edi
    and    edi, DWORD PTR _a5$[esp+32]
    mov    ecx, edx
    not    ecx
    and    ebp, esi
    mov    eax, ecx
    and    eax, esi
    and    ecx, ebx
    mov    DWORD PTR _x1$[esp+36], eax
```

```
    xor     eax, ebx
    mov     esi, ebp
    or      esi, edx
    mov     DWORD PTR _x4$[esp+36], esi
    and     esi, DWORD PTR _a6$[esp+32]
    mov     DWORD PTR _x7$[esp+32], ecx
    mov     edx, esi
    xor     edx, eax
    mov     DWORD PTR _x6$[esp+36], edx
    mov     edx, DWORD PTR _a3$[esp+32]
    xor     edx, ebx
    mov     ebx, esi
    xor     ebx, DWORD PTR _a5$[esp+32]
    mov     DWORD PTR _x8$[esp+36], edx
    and     ebx, edx
    mov     ecx, edx
    mov     edx, ebx
    xor     edx, ebp
    or      edx, DWORD PTR _a6$[esp+32]
    not     ecx
    and     ecx, DWORD PTR _a6$[esp+32]
    xor     edx, edi
    mov     edi, edx
    or      edi, DWORD PTR _a2$[esp+32]
    mov     DWORD PTR _x3$[esp+36], ebp
    mov     ebp, DWORD PTR _a2$[esp+32]
    xor     edi, ebx
    and     edi, DWORD PTR _a1$[esp+32]
    mov     ebx, ecx
    xor     ebx, DWORD PTR _x7$[esp+32]
    not     edi
    or      ebx, ebp
    xor     edi, ebx
    mov     ebx, edi
    mov     edi, DWORD PTR _out2$[esp+32]
    xor     ebx, DWORD PTR [edi]
    not     eax
    xor     ebx, DWORD PTR _x6$[esp+36]
    and     eax, edx
    mov     DWORD PTR [edi], ebx
    mov     ebx, DWORD PTR _x7$[esp+32]
    or      ebx, DWORD PTR _x6$[esp+36]
    mov     edi, esi
    or      edi, DWORD PTR _x1$[esp+36]
    mov     DWORD PTR _x28$[esp+32], ebx
    xor     edi, DWORD PTR _x8$[esp+36]
    mov     DWORD PTR _x24$[esp+32], edi
    xor     edi, ecx
    not     edi
    and     edi, edx
    mov     ebx, edi
    and     ebx, ebp
    xor     ebx, DWORD PTR _x28$[esp+32]
```

```
    xor     ebx, eax
    not     eax
    mov     DWORD PTR _x33$[esp+32], ebx
    and     ebx, DWORD PTR _a1$[esp+32]
    and     eax, ebp
    xor     eax, ebx
    mov     ebx, DWORD PTR _out4$[esp+32]
    xor     eax, DWORD PTR [ebx]
    xor     eax, DWORD PTR _x24$[esp+32]
    mov     DWORD PTR [ebx], eax
    mov     eax, DWORD PTR _x28$[esp+32]
    and     eax, DWORD PTR _a3$[esp+32]
    mov     ebx, DWORD PTR _x3$[esp+36]
    or      edi, DWORD PTR _a3$[esp+32]
    mov     DWORD PTR _x36$[esp+32], eax
    not     eax
    and     eax, edx
    or      ebx, ebp
    xor     ebx, eax
    not     eax
    and     eax, DWORD PTR _x24$[esp+32]
    not     ebp
    or      eax, DWORD PTR _x3$[esp+36]
    not     esi
    and     ebp, eax
    or      eax, edx
    xor     eax, DWORD PTR _a5$[esp+32]
    mov     edx, DWORD PTR _x36$[esp+32]
    xor     edx, DWORD PTR _x4$[esp+36]
    xor     ebp, edi
    mov     edi, DWORD PTR _out1$[esp+32]
    not     eax
    and     eax, DWORD PTR _a2$[esp+32]
    not     ebp
    and     ebp, DWORD PTR _a1$[esp+32]
    and     edx, esi
    xor     eax, edx
    or      eax, DWORD PTR _a1$[esp+32]
    not     ebp
    xor     ebp, DWORD PTR [edi]
    not     ecx
    and     ecx, DWORD PTR _x33$[esp+32]
    xor     ebp, ebx
    not     eax
    mov     DWORD PTR [edi], ebp
    xor     eax, ecx
    mov     ecx, DWORD PTR _out3$[esp+32]
    xor     eax, DWORD PTR [ecx]
    pop     edi
    pop     esi
    xor     eax, ebx
    pop     ebp
    mov     DWORD PTR [ecx], eax
```

```
    pop    ebx
    add    esp, 20
    ret    0
_s1    ENDP
```

Seules 5 variables ont été allouées dans la pile par le compilateur.

Essayons maintenant une compilation avec la version 64 bits de MSVC 2008:

Listing 1.399 : MSVC 2008 avec optimisation

```
a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$1$ = 88
a5$ = 88
a6$ = 96
out1$ = 104
out2$ = 112
out3$ = 120
out4$ = 128
s1    PROC
$LN3:
    mov    QWORD PTR [rsp+24], rbx
    mov    QWORD PTR [rsp+32], rbp
    mov    QWORD PTR [rsp+16], rdx
    mov    QWORD PTR [rsp+8], rcx
    push   rsi
    push   rdi
    push   r12
    push   r13
    push   r14
    push   r15
    mov    r15, QWORD PTR a5$[rsp]
    mov    rcx, QWORD PTR a6$[rsp]
    mov    rbp, r8
    mov    r10, r9
    mov    rax, r15
    mov    rdx, rbp
    not    rax
    xor    rdx, r9
    not    r10
    mov    r11, rax
    and    rax, r9
    mov    rsi, r10
    mov    QWORD PTR x36$1$[rsp], rax
    and    r11, r8
    and    rsi, r8
    and    r10, r15
    mov    r13, rdx
    mov    rbx, r11
    xor    rbx, r9
    mov    r9, QWORD PTR a2$[rsp]
```

```
    mov     r12, rsi
    or      r12, r15
    not     r13
    and     r13, rcx
    mov     r14, r12
    and     r14, rcx
    mov     rax, r14
    mov     r8, r14
    xor     r8, rbx
    xor     rax, r15
    not     rbx
    and     rax, rdx
    mov     rdi, rax
    xor     rdi, rsi
    or      rdi, rcx
    xor     rdi, r10
    and     rbx, rdi
    mov     rcx, rdi
    or      rcx, r9
    xor     rcx, rax
    mov     rax, r13
    xor     rax, QWORD PTR x36$1$[rsp]
    and     rcx, QWORD PTR a1$[rsp]
    or      rax, r9
    not     rcx
    xor     rcx, rax
    mov     rax, QWORD PTR out2$[rsp]
    xor     rcx, QWORD PTR [rax]
    xor     rcx, r8
    mov     QWORD PTR [rax], rcx
    mov     rax, QWORD PTR x36$1$[rsp]
    mov     rcx, r14
    or      rax, r8
    or      rcx, r11
    mov     r11, r9
    xor     rcx, rdx
    mov     QWORD PTR x36$1$[rsp], rax
    mov     r8, rsi
    mov     rdx, rcx
    xor     rdx, r13
    not     rdx
    and     rdx, rdi
    mov     r10, rdx
    and     r10, r9
    xor     r10, rax
    xor     r10, rbx
    not     rbx
    and     rbx, r9
    mov     rax, r10
    and     rax, QWORD PTR a1$[rsp]
    xor     rbx, rax
    mov     rax, QWORD PTR out4$[rsp]
    xor     rbx, QWORD PTR [rax]
```

```
    xor     rbx, rcx
    mov     QWORD PTR [rax], rbx
    mov     rbx, QWORD PTR x36$1$[rsp]
    and     rbx, rbp
    mov     r9, rbx
    not     r9
    and     r9, rdi
    or      r8, r11
    mov     rax, QWORD PTR out1$[rsp]
    xor     r8, r9
    not     r9
    and     r9, rcx
    or      rdx, rbp
    mov     rbp, QWORD PTR [rsp+80]
    or      r9, rsi
    xor     rbx, r12
    mov     rcx, r11
    not     rcx
    not     r14
    not     r13
    and     rcx, r9
    or      r9, rdi
    and     rbx, r14
    xor     r9, r15
    xor     rcx, rdx
    mov     rdx, QWORD PTR a1$[rsp]
    not     r9
    not     rcx
    and     r13, r10
    and     r9, r11
    and     rcx, rdx
    xor     r9, rbx
    mov     rbx, QWORD PTR [rsp+72]
    not     rcx
    xor     rcx, QWORD PTR [rax]
    or      r9, rdx
    not     r9
    xor     rcx, r8
    mov     QWORD PTR [rax], rcx
    mov     rax, QWORD PTR out3$[rsp]
    xor     r9, r13
    xor     r9, QWORD PTR [rax]
    xor     r9, r8
    mov     QWORD PTR [rax], r9
    pop     r15
    pop     r14
    pop     r13
    pop     r12
    pop     rdi
    pop     rsi
    ret     0
s1    ENDP
```

Le compilateur n'a pas eu besoin d'allouer de l'espace sur la pile. x36 est synonyme de a5.

Il existe cependant des CPUs qui possèdent beaucoup plus de GPR. Itanium possède ainsi 128 registres.

1.37.2 ARM

Les instructions 64 bits sont apparues avec ARMv8.

1.37.3 Nombres flottants

Le traitement des nombres flottants en environnement x86-64 est expliqué ici: 1.38 on the following page.

1.37.4 Critiques concernant l'architecture 64 bits

Certains se sont parfois irrité du doublement de la taille des pointeurs, en particulier dans le cache puisque les CPUs x64 ne peuvent de toute manière adresser que des adresses RAM sur 48 bits.

> Les pointeurs ont perdu mes faveurs au point que j'en viens à les injurier. Si je cherche vraiment à utiliser au mieux les capacités de mon ordinateur 64 bits, j'en conclus que je ferais mieux de ne pas utiliser de pointeurs. Les registres de mon ordinateur sont sur 64 bits, mais je n'ai que 2Go de RAM. Les pointeurs n'ont donc jamais plus de 32 bits significatifs. Et pourtant, chaque fois que j'utilise un pointeur, il me coûte 64 bits ce qui double la taille de ma structure de données. Pire, ils atterrissent dans mon cache et en gaspillent la moitié et cela me coûte car le cache est cher.
>
> Donc, si je cherche à grappiller, j'en viens à utiliser des tableaux au lieu de pointeurs. Je rédige des macros compliquées qui peuvent laisser l'impression à tort que j'utilise des pointeurs.

(Donald Knuth dans "Coders at Work: Reflections on the Craft of Programming ".)

Certains en sont venus à fabriquer leurs propres allocateurs de mémoire.

Il est intéressant de se pencher sur le cas de CryptoMiniSat[182]. Ce programme qui utilise rarement plus de 4Go de RAM, fait un usage intensif des pointeurs. Il nécessite donc moins de mémoire sur les architectures 32 bits que sur celles à 64 bits. Pour remédier à ce problème, l'auteur a donc programmé son propre allocateur (dans *clauseallocator.(h|cpp)*), qui lui permet d'allouer de la mémoire en utilisant des identifiants sur 32 bits à la place de pointeurs sur 64 bits.

[182] `https://github.com/msoos/cryptominisat/`

1.38 Travailler avec des nombres à virgule flottante en utilisant SIMD

Bien sûr. le FPU est resté dans les processeurs compatible x86 lorsque les extensions SIMD ont été ajoutées.

L'extension SIMD (SSE2) offre un moyen facile de travailler avec des nombres à virgule flottante.

Le format des nombres reste le même (IEEE 754).

Donc, les compilateurs modernes (incluant ceux générant pour x86-64) utilisent les instructions SIMD au lieu de celles pour FPU.

On peut dire que c'est une bonne nouvelle, car il est plus facile de travailler avec elles.

Nous allons ré-utiliser les exemples de la section FPU ici: 1.25 on page 285.

1.38.1 Simple exemple

```
#include <stdio.h>

double f (double a, double b)
{
        return a/3.14 + b*4.1;
};

int main()
{
        printf ("%f\n", f(1.2, 3.4));
};
```

x64

Listing 1.400 : MSVC 2012 x64 avec optimisation

```
__real@4010666666666666 DQ 04010666666666666r   ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr   ; 3.14

a$ = 8
b$ = 16
f       PROC
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        ret     0
f       ENDP
```

Les valeurs en virgule flottante entrées sont passées dans les registres XMM0-XMM3, tout le reste—via la pile [183].

[183] MSDN: Parameter Passing

a est passé dans XMM0, b—via XMM1.

Les registres XMM font 128-bit (comme nous le savons depuis la section à propos de SIMD : 1.36 on page 525), mais les valeurs *double* font 64-bit, donc seulement la moitié basse du registre est utilisée.

DIVSD est une instruction SSE qui signifie «Divide Scalar Double-Precision Floating-Point Values» (Diviser des nombres flottants double-précision), elle divise une valeur de type *double* par une autre, stockées dans la moitié basse des opérandes.

Les constantes sont encodées par le compilateur au format IEEE 754.

MULSD et ADDSD fonctionnent de même, mais font la multiplication et l'addition.

Le résultat de l'exécution de la fonction de type *double* est laissé dans le registre XMM0.

C'est ainsi que travaille MSVC sans optimisation:

Listing 1.401 : MSVC 2012 x64

```
__real@4010666666666666 DQ 04010666666666666r   ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr   ; 3.14

a$ = 8
b$ = 16
f       PROC
        movsdx  QWORD PTR [rsp+16], xmm1
        movsdx  QWORD PTR [rsp+8], xmm0
        movsdx  xmm0, QWORD PTR a$[rsp]
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        movsdx  xmm1, QWORD PTR b$[rsp]
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        ret     0
f       ENDP
```

Légèrement redondant. Les arguments en entrée sont sauvés dans le «shadow space» (1.14.2 on page 137), mais seule leur moitié inférieure, i.e., seulement la valeur 64-bit de type *double*. GCC produit le même code.

x86

Compilons cet exemple pour x86. Bien qu'il compile pour x86, MSVC 2012 utilise des instructions SSE2:

Listing 1.402 : MSVC 2012 x86 sans optimisation

```
tv70 = -8           ; size = 8
_a$ = 8             ; size = 8
_b$ = 16            ; size = 8
_f      PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 8
```

```
        movsd   xmm0, QWORD PTR _a$[ebp]
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        movsd   xmm1, QWORD PTR _b$[ebp]
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        movsd   QWORD PTR tv70[ebp], xmm0
        fld     QWORD PTR tv70[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

Listing 1.403 : MSVC 2012 x86 avec optimisation

```
tv67 = 8          ; size = 8
_a$ = 8           ; size = 8
_b$ = 16          ; size = 8
_f      PROC
        movsd   xmm1, QWORD PTR _a$[esp-4]
        divsd   xmm1, QWORD PTR __real@40091eb851eb851f
        movsd   xmm0, QWORD PTR _b$[esp-4]
        mulsd   xmm0, QWORD PTR __real@4010666666666666
        addsd   xmm1, xmm0
        movsd   QWORD PTR tv67[esp-4], xmm1
        fld     QWORD PTR tv67[esp-4]
        ret     0
_f      ENDP
```

C'est presque le même code, toutefois, il y a quelques différences relatives aux conventions d'appel: 1) les arguments ne sont pas passés dans des registres XMM, mais par la pile, comme dans les exemples FPU (1.25 on page 285); 2) le résultat de la fonction est renvoyé dans ST(0) — afin de faire cela, il est copié (à travers la variable locale tv) depuis un des registres XMM dans ST(0).

Essayons l'exemple optimisé dans OllyDbg :

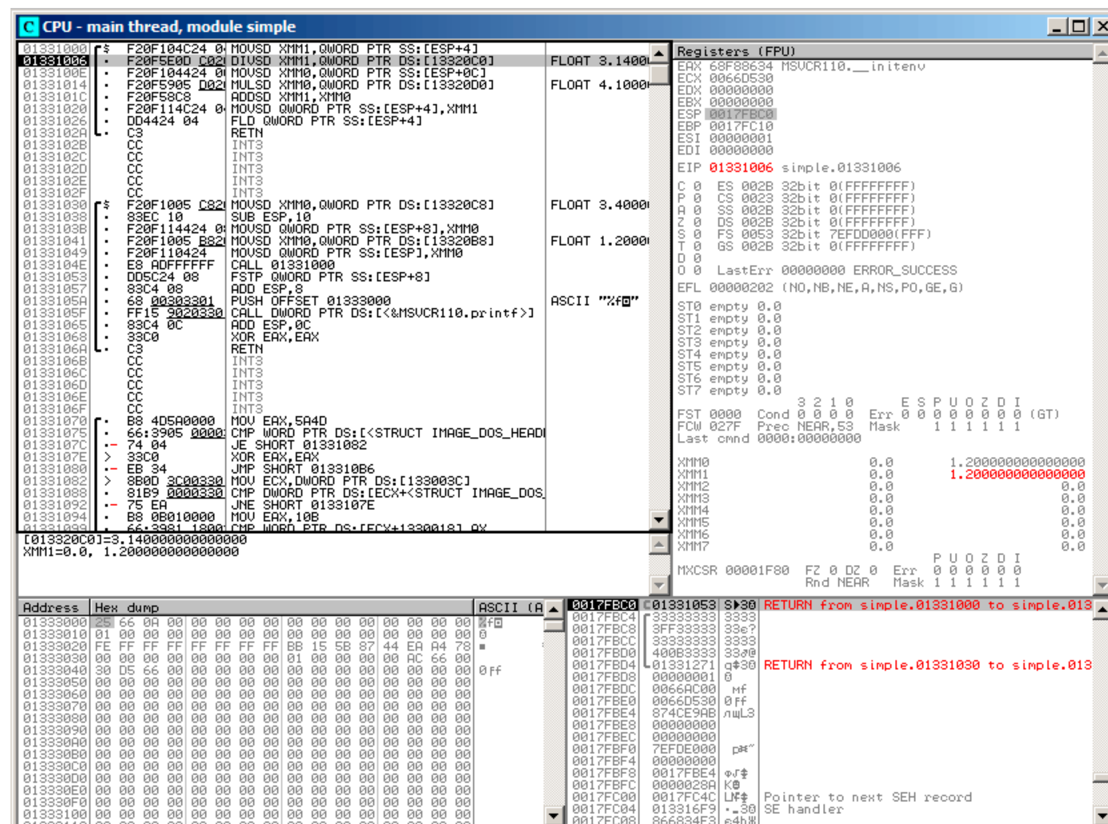


Fig. 1.113: OllyDbg : MOVSD charge la valeur de a dans XMM1

Fig. 1.114: OllyDbg : DIVSD a calculé le quotient et l'a stocké dans XMM1

Fig. 1.115: OllyDbg : MULSD a calculé le produit et l'a stocké dans XMM0

Fig. 1.116: OllyDbg : ADDSD ajoute la valeur dans XMM0 à celle dans XMM1

Fig. 1.117: OllyDbg : FLD laisse le résultat de la fonction dans `ST(0)`

Nous voyons qu'OllyDbg montre les registres XMM comme des paires de nombres *double*, mais seule la partie *basse* est utilisée.

Apparemment, OllyDbg les montre dans ce format car les instructions SSE2 (suffixées avec `-SD`) sont exécutées actuellement.

Mais bien sûr, il est possible de changer le format du registre et de voir le contenu comme 4 nombres *float* ou juste comme 16 octets.

1.38.2 Passer des nombres à virgule flottante via les arguments

```
#include <math.h>
#include <stdio.h>

int main ()
{
        printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

        return 0;
}
```

Ils sont passés dans la moitié basse des registres XMM0-XMM3.

Listing 1.404 : MSVC 2012 x64 avec optimisation

```
$SG1354 DB      '32.01 ^ 1.54 = %lf', 0aH, 00H

__real@40400147ae147ae1 DQ 040400147ae147ae1r   ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r   ; 1.54

main    PROC
        sub     rsp, 40                                 ; 00000028H
        movsdx  xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
        movsdx  xmm0, QWORD PTR __real@40400147ae147ae1
        call    pow
        lea     rcx, OFFSET FLAT:$SG1354
        movaps  xmm1, xmm0
        movd    rdx, xmm1
        call    printf
        xor     eax, eax
        add     rsp, 40                                 ; 00000028H
        ret     0
main    ENDP
```

Il n'y a pas d'instruction MOVSDX dans les manuels Intel et AMD (12.1.4 on page 1315), elle y est appelée MOVSD. Donc il y a deux instructions qui partagent le même nom en x86 (à propos de l'autre lire: .1.6 on page 1334). Apparemment, les développeurs de Microsoft voulaient arrêter cette pagaille, donc ils l'ont renommée MOVSDX. Elle charge simplement une valeur dans la moitié inférieure d'un registre XMM.

pow() prends ses arguments de XMM0 et XMM1, et renvoie le résultat dans XMM0. Il est ensuite déplacé dans RDX pour printf(). Pourquoi? Peut-être parce que printf()— est une fonction avec un nombre variable d'arguments?

Listing 1.405 : GCC 4.4.6 x64 avec optimisation

```
.LC2:
        .string "32.01 ^ 1.54 = %lf\n"
main:
        sub     rsp, 8
        movsd   xmm1, QWORD PTR .LC0[rip]
        movsd   xmm0, QWORD PTR .LC1[rip]
```

```
        call    pow
        ; le résultat est maintenant dans XMM0
        mov     edi, OFFSET FLAT:.LC2
        mov     eax, 1 ; nombre de registres vecteur passé
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
.LC0:
        .long   171798692
        .long   1073259479
.LC1:
        .long   2920577761
        .long   1077936455
```

GCC génère une sortie plus claire. La valeur pour `printf()` est passée dans XMM0. À propos, il y a un cas lorsque 1 est écrit dans EAX pour `printf()`—ceci implique qu'un argument sera passé dans des registres vectoriels, comme le requiert le standard [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [184].

1.38.3 Exemple de comparaison

```
#include <stdio.h>

double d_max (double a, double b)
{
        if (a>b)
                return a;

        return b;
};

int main()
{
        printf ("%f\n", d_max (1.2, 3.4));
        printf ("%f\n", d_max (5.6, -4));
};
```

x64

Listing 1.406 : MSVC 2012 x64 avec optimisation

```
a$ = 8
b$ = 16
d_max   PROC
        comisd  xmm0, xmm1
        ja      SHORT $LN2@d_max
```

[184] Aussi disponible en https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

```
        movaps  xmm0, xmm1
$LN2@d_max:
        fatret  0
d_max   ENDP
```

MSVC avec optimisation génère un code très facile à comprendre.

`COMISD` is «Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS» (comparer des valeurs double précision en virgule flottante scalaire ordrées et mettre les EFLAGS). Pratiquement, c'est ce qu'elle fait.

MSVC sans optimisation génère plus de code redondant, mais il n'est toujours pas très difficile à comprendre:

Listing 1.407 : MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max   PROC
        movsdx  QWORD PTR [rsp+16], xmm1
        movsdx  QWORD PTR [rsp+8], xmm0
        movsdx  xmm0, QWORD PTR a$[rsp]
        comisd  xmm0, QWORD PTR b$[rsp]
        jbe     SHORT $LN1@d_max
        movsdx  xmm0, QWORD PTR a$[rsp]
        jmp     SHORT $LN2@d_max
$LN1@d_max:
        movsdx  xmm0, QWORD PTR b$[rsp]
$LN2@d_max:
        fatret  0
d_max   ENDP
```

Toutefois, GCC 4.4.6 effectue plus d'optimisations et utilise l'instruction `MAXSD` («Return Maximum Scalar Double-Precision Floating-Point Value») qui choisit la valeur maximum!

Listing 1.408 : GCC 4.4.6 x64 avec optimisation

```
d_max:
        maxsd   xmm0, xmm1
        ret
```

x86

Compilons cet exemple dans MSVC 2012 avec l'optimisation activée:

Listing 1.409 : MSVC 2012 x86 avec optimisation

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_d_max  PROC
        movsd   xmm0, QWORD PTR _a$[esp-4]
        comisd  xmm0, QWORD PTR _b$[esp-4]
        jbe     SHORT $LN1@d_max
        fld     QWORD PTR _a$[esp-4]
        ret     0
$LN1@d_max:
        fld     QWORD PTR _b$[esp-4]
        ret     0
_d_max  ENDP
```

Presque la même chose, mais les valeurs de a et b sont prises depuis la pile et le résultat de la fonction est laissé dans `ST(0)`.

Si nous chargeons cet exemple dans OllyDbg, nous pouvons voir comment l'instruction `COMISD` compare les valeurs et met/efface les flags `CF` et `PF` :

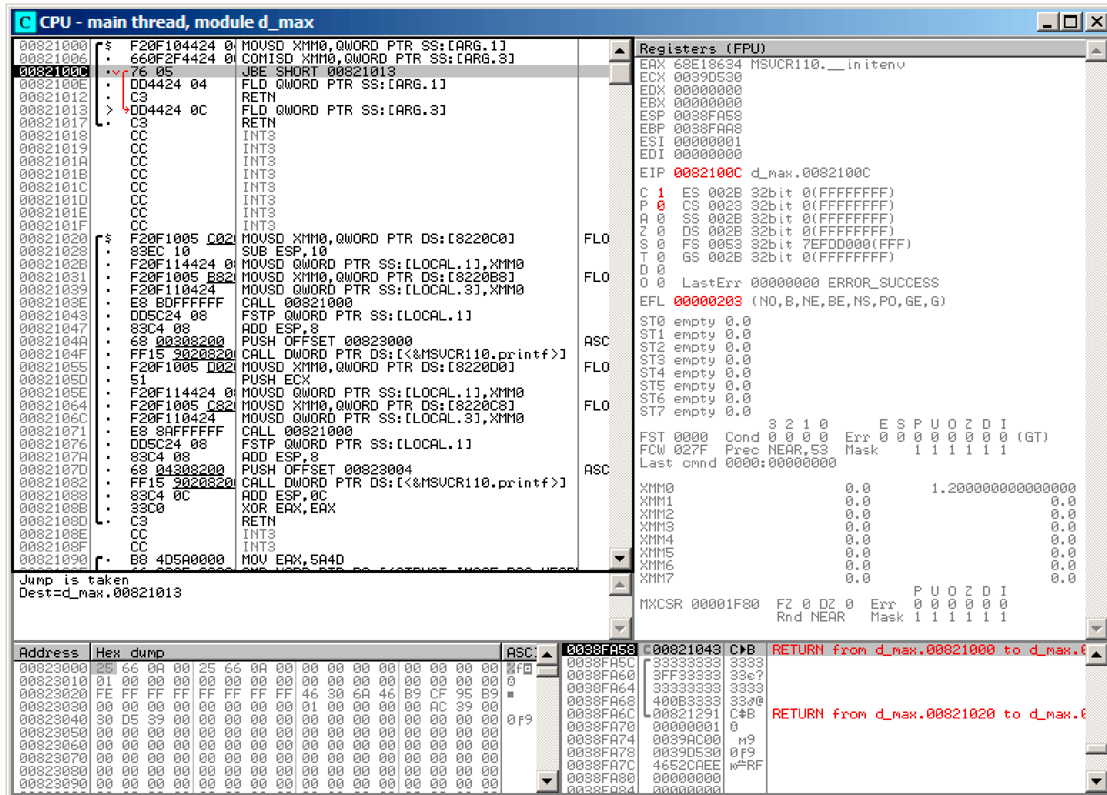


Fig. 1.118: OllyDbg : COMISD a changé les flags CF et PF

1.38.4 Calcul de l'epsilon de la machine: x64 et SIMD

Revoyons l'exemple «calcul de l'epsilon de la machine» pour *double* listado.1.32.2.

Maintenant nous compilons pour x64:

Listing 1.410 : MSVC 2012 x64 avec optimisation

```
v$ = 8
calculate_machine_epsilon PROC
        movsdx  QWORD PTR v$[rsp], xmm0
        movaps  xmm1, xmm0
        inc     QWORD PTR v$[rsp]
        movsdx  xmm0, QWORD PTR v$[rsp]
        subsd   xmm0, xmm1
        ret     0
calculate_machine_epsilon ENDP
```

Il n'y a pas moyen d'ajouter 1 à une valeur dans un registre XMM 128-bit, donc il doit être placé en mémoire.

Il y a toutefois l'instruction ADDSD (*Add Scalar Double-Precision Floating-Point Values*

ajouter des valeurs scalaires à virgule flottante double-précision), qui peut ajouter une valeur dans la moitié 64-bit basse d'un registre XMM en ignorant celle du haut, mais MSVC 2012 n'est probablement pas encore assez bon [185].

Néanmoins, la valeur est ensuite rechargée dans un registre XMM et la soustraction est effectuée. SUBSD est «Subtract Scalar Double-Precision Floating-Point Values» (soustraire des valeurs en virgule flottante double-précision), i.e., elle opère sur la partie 64-bit basse d'un registre XMM 128-bit. Le résultat est renvoyé dans le registre XMM0.

1.38.5 Exemple de générateur de nombre pseudo-aléatoire revisité

Revoyons l'exemple de «générateur de nombre pseudo-aléatoire» listado.1.32.1.

Si nous compilons ceci en MSVC 2012, il va utiliser les instructions SIMD pour le FPU.

Listing 1.411 : MSVC 2012 avec optimisation

```
__real@3f800000 DD 03f800000r   ; 1

tv128 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
        push    ecx
        call    ?my_rand@@YAIXZ
; EAX=valeur pseudo-aléatoire
        and     eax, 8388607    ; 007fffffH
        or      eax, 1065353216 ; 3f800000H
; EAX=valeur pseudo-aléatoire & 0x007fffff | 0x3f800000
; la stocker dans la pile locale:
        mov     DWORD PTR _tmp$[esp+4], eax
; la recharger comme un nombre à virgule flottante:
        movss   xmm0, DWORD PTR _tmp$[esp+4]
; soustraire 1.0:
        subss   xmm0, DWORD PTR __real@3f800000
; mettre la valeur dans ST0 en la plaçant dans une variable temporaire...
        movss   DWORD PTR tv128[esp+4], xmm0
; ... et en la rechargeant dans ST0:
        fld     DWORD PTR tv128[esp+4]
        pop     ecx
        ret     0
?float_rand@@YAMXZ ENDP
```

Toutes les instructions ont le suffixe -SS, qui signifie «Scalar Single» (scalaire simple).

«Scalar» (scalaire) implique qu'une seule valeur est stockée dans le registre.

«Single» (simple[186]) signifie un type de donnée *float*.

[185] À titre d'exercice, vous pouvez retravailler ce code pour éliminer l'usage de la pile locale
[186] pour simple précision

1.38.6 Résumé

Seule la moitié basse des registres XMM est utilisée dans tous les exemples ici, pour stocker un nombre au format IEEE 754.

Pratiquement, toutes les instructions préfixées par `-SD` («Scalar Double-Precision »)— sont des instructions travaillant avec des nombres à virgule flottante au format IEEE 754, stockés dans la moitié 64-bit basse d'un registre XMM.

Et c'est plus facile que dans le FPU, sans doute parce que les extensions SIMD ont évolué dans un chemin moins chaotique que celles FPU dans le passé. Le modèle de pile de registre n'est pas utilisé.

Si vous voulez, essayez de remplacer *double* avec *float*

dans ces exemples, la même instruction sera utilisée, mais préfixée avec `-SS` («Scalar Single-Precision » scalaire simple-précision), par exemple, `MOVSS`, `COMISS`, `ADDSS`, etc.

«Scalaire » implique que le registre SIMD contienne seulement une valeur au lieu de plusieurs.

Les instructions travaillant avec plusieurs valeurs dans un registre simultanément ont «Packed » dans leur nom.

Inutile de dire que les instructions SSE2 travaillent avec des nombres 64-bit au format IEEE 754 (*double*), alors que la représentation interne des nombres à virgule flottante dans le FPU est sur 80-bit.

C'est pourquoi la FPU produit moins d'erreur d'arrondi et par conséquent, le FPU peut donner des résultats de calcul plus précis.

1.39 Détails spécifiques à ARM

1.39.1 Signe (#) avant un nombre

Le compilateur Keil, IDA et objdump font précéder tous les nombres avec le signe « # », par exemple: listado.1.22.1.

Mais lorsque GCC 4.9 génère une sortie en langage d'assemblage, il ne le fait pas, par exemple: listado.3.18.

Les listings ARM dans ce livre sont quelque peu mélangés.

Il est difficile de dire quelle méthode est juste. On est supposé suivre les règles admises de l'environnement dans lequel on travaille.

1.39.2 Modes d'adressage

Cette instruction est possible en ARM64:

```
ldr     x0, [x29,24]
```

Ceci signifie ajouter 24 à la valeur dans X29 et charger la valeur à cette adresse.

Notez s'il vous plaît que 24 est à l'intérieur des parenthèses. La signification est différente si le nombre est à l'extérieur des parenthèses:

```
ldr     w4, [x1],28
```

Ceci signifie charger la valeur à l'adresse dans X1, puis ajouter 28 à X1.

ARM permet d'ajouter ou de soustraire une constante à/de l'adresse utilisée pour charger.

Et il est possible de faire cela à la fois avant et après le chargement.

Il n'y a pas de tels modes d'adressage en x86, mais ils sont présents dans d'autres processeurs, même sur le PDP-11.

Il y a une légende disant que les modes pré-incrémentation, post-incrémentation, pré-décrémentation et post-décrémentation du PDP-11, sont «responsables» de l'apparition du genre de constructions en langage C (qui a été développé sur PDP-11) comme *ptr++, *++ptr, *ptr--, *--ptr.

À propos, ce sont des caractéristiques de C difficiles à mémoriser. Voici comment ça se passe:

terme C	terme ARM	déclaration C	ce que ça fait
Post-incrémentation	adressage post-indexé	`*ptr++`	utiliser la valeur `*ptr`, puis incrémenter le pointeur `ptr`
Post-décrémentation	adressage post-indexé	`*ptr--`	utiliser la valeur `*ptr`, puis décrémenter le pointeur `ptr`
Pré-incrémentation	adressage pré-indexé	`*++ptr`	incrémenter le pointeur `ptr`, puis utiliser la valeur `*ptr`
Pré-décrémentation	adressage pré-indexé	`*--ptr`	décrémenter le pointeur `ptr`, puis utiliser la valeur `*ptr`

La pré-indexation est marquée avec un point d'exclamation en langage d'assemblage ARM. Par exemple, voir ligne 2 dans listado.1.28.

Dennis Ritchie (un des créateurs du langage C) a mentionné que cela a vraisemblablement été inventé par Ken Thompson (un autre créateur du C) car cette possibilité était présente sur le PDP-7 [187], [Dennis M. Ritchie, *The development of the C language*, (1993)][188].

Ainsi, les compilateurs de langage C peuvent l'utiliser, si elle est présente sur le processeur cible.

C'est très pratique pour le traitement de tableau.

[187] `http://yurichev.com/mirrors/C/c_dmr_postincrement.txt`
[188] Aussi disponible en pdf

1.39.3 Charger une constante dans un registre

ARM 32-bit

Comme nous le savons déjà, toutes les instructions ont une taille de 4 octets en mode ARM et de 2 octets en mode Thumb.

Mais comment peut-on charger une valeur 32-bit dans un registre, s'il n'est pas possible de l'encoder dans une instruction?

Essayons:

```
unsigned int f()
{
        return 0x12345678;
};
```

Listing 1.412 : GCC 4.6.3 -O3 Mode ARM

```
f:
        ldr     r0, .L2
        bx      lr
.L2:
        .word   305419896 ; 0x12345678
```

Donc, la valeur `0x12345678` est simplement stockée à part en mémoire et chargée si besoin.

Mais il est possible de se débarrasser de l'accès supplémentaire en mémoire.

Listing 1.413 : GCC 4.6.3 -O3 -march=armv7-a (Mode ARM)

```
movw    r0, #22136      ; 0x5678
movt    r0, #4660       ; 0x1234
bx      lr
```

Nous voyons que la valeur est chargée dans le registre par parties, la partie basse en premier (en utilisant `MOVW`), puis la partie haute (en utilisant `MOVT`).

Ceci implique que 2 instructions sont nécessaires en mode ARM pour charger une valeur 32-bit dans un registre.

Ce n'est pas un problème, car en fait il n'y pas beaucoup de constantes dans du code réel (excepté pour 0 et 1).

Est-ce que ça signifie que la version à deux instructions est plus lente que celle à une instruction?

C'est discutable. Le plus souvent, les processeurs ARM modernes sont capable de détecter de telle séquences et les exécutent rapidement.

D'un autre côté, IDA est capable de détecter ce genre de patterns dans le code et désassemble cette fonction comme:

```
MOV    R0, 0x12345678
BX     LR
```

ARM64

```
uint64_t f()
{
        return 0x12345678ABCDEF01;
};
```

Listing 1.414 : GCC 4.9.1 -O3

```
mov     x0, 61185   ; 0xef01
movk    x0, 0xabcd, lsl 16
movk    x0, 0x5678, lsl 32
movk    x0, 0x1234, lsl 48
ret
```

MOVK signifie «MOV Keep» (déplacer garder), i.e., elle écrit une valeur 16-bit dans le registre, sans affecter le reste des bits. Le suffixe LSL signifie décaler la valeur à gauche de 16, 32 et 48 bits à chaque étape. Le décalage est fait avant le chargement.

Ceci implique que 4 instructions sont nécessaires pour charger une valeur de 64-bit dans un registre.

Charger un nombre à virgule flottante dans un registre

Il est possible de stocker un nombre à virgule flottante dans un D-registre en utilisant une seule instruction.

Par exemple:

```
double a()
{
        return 1.5;
};
```

Listing 1.415 : GCC 4.9.1 -O3 + objdump

```
0000000000000000 <a>:
   0:   1e6f1000        fmov    d0, #1.500000000000000000e+000
   4:   d65f03c0        ret
```

Le nombre 1.5 a en effet été encodé dans une instruction 32-bit. Mais comment?

En ARM64, il y a 8 bits dans l'instruction FMOV pour encoder certains nombres à virgule flottante.

L'algorithme est appelé VFPExpandImm() en [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)][189]. Ceci est aussi appelé *minifloat*[190] (mini flottant).

[189]Aussi disponible en http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf
[190]Wikipédia

Nous pouvons essayer différentes valeurs. Le compilateur est capable d'encoder 30.0 et 31.0, mais il ne peut pas encoder 32.0, car 8 octets doivent être alloués pour ce nombre au format IEEE 754:

```
double a()
{
        return 32;
};
```

Listing 1.416 : GCC 4.9.1 -O3

```
a:
        ldr     d0, .LC0
        ret
.LC0:
        .word   0
        .word   1077936128
```

1.39.4 Relogement en ARM64

Comme nous le savons, il y a des instructions 4-octet en ARM64, donc il est impossible d'écrire un nombre large dans un registre en utilisant une seule instruction.

Cependant, une image exécutable peut être chargée à n'importe quelle adresse aléatoire en mémoire, c'est pourquoi les relogements existent.

L'adresse est formée en utilisant la paire d'instructions ADRP et ADD en ARM64.

La première charge l'adresse d'une page de 4KiB et la seconde ajoute le reste. Compilons l'exemple de «Hello, world! » (listado.1.11) avec GCC (Linaro) 4.9 sous win32:

Listing 1.417 : GCC (Linaro) 4.9 et objdump du fichier objet

```
...>aarch64-linux-gnu-gcc.exe hw.c -c

...>aarch64-linux-gnu-objdump.exe -d hw.o

...

0000000000000000 <main>:
   0:   a9bf7bfd        stp     x29, x30, [sp,#-16]!
   4:   910003fd        mov     x29, sp
   8:   90000000        adrp    x0, 0 <main>
   c:   91000000        add     x0, x0, #0x0
  10:   94000000        bl      0 <printf>
  14:   52800000        mov     w0, #0x0    // #0
  18:   a8c17bfd        ldp     x29, x30, [sp],#16
  1c:   d65f03c0        ret

...>aarch64-linux-gnu-objdump.exe -r hw.o

...

RELOCATION RECORDS FOR [.text]:
```

```
OFFSET           TYPE              VALUE
0000000000000008 R_AARCH64_ADR_PREL_PG_HI21  .rodata
000000000000000c R_AARCH64_ADD_ABS_LO12_NC  .rodata
0000000000000010 R_AARCH64_CALL26  printf
```

Donc, il y a 3 relogements dans ce fichier objet.

- La première prend l'adresse de la page, coupe les 12 bits du bas et écrit les 21 bits du haut restants dans le champs de bit de l'instruction ADRP. Ceci car nous n'avons pas besoin d'encoder les 12 bits bas, et l'instruction ADRP possède seulement de l'espace pour 21 bits.
- La seconde met les 12 bits de l'adresse relative au début de la page dans le champ de bits de l'instruction ADD.
- La dernière, celle de 26-bit, est appliquée à l'instruction à l'adresse `0x10` où le saut à la fonction `printf()` se trouve.

  Toutes les adresses des instructions ARM64 (et ARM en mode ARM) ont zéro dans les deux bits les plus bas (car toutes les instructions ont une taille de 4 octets), donc on doit seulement encoder les 26 bits du haut de l'espace d'adresse de 28-bit (± 128MB).

Il n'y a pas de tels relogements dans le fichier exécutable: car l'adresse où se trouve la chaîne «Hello! » est connue, la page, et l'adresse de `puts()` sont aussi connues.

Donc, il y a déjà des valeurs mises dans les instructions ADRP, ADD et BL (l'éditeur de liens à écrit des valeurs lors de l'édition de liens) :

Listing 1.418 : objdump du fichier exécutable

```
0000000000400590 <main>:
  400590:       a9bf7bfd        stp     x29, x30, [sp,#-16]!
  400594:       910003fd        mov     x29, sp
  400598:       90000000        adrp    x0, 400000 <_init-0x3b8>
  40059c:       91192000        add     x0, x0, #0x648
  4005a0:       97ffffa0        bl      400420 <puts@plt>
  4005a4:       52800000        mov     w0, #0x0  // #0
  4005a8:       a8c17bfd        ldp     x29, x30, [sp],#16
  4005ac:       d65f03c0        ret

...

Contents of section .rodata:
 400640 01000200 00000000 48656c6c 6f210000  ........Hello!..
```

À titre d'exemple, essayons de désassembler manuellement l'instruction BL.
`0x97ffffa0` est $0b10010111111111111111111110100000$. D'après [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)C5.6.26], *imm26* correspond aux derniers 26 bits:
$imm26 = 0b11111111111111111110100000$. Il s'agit de `0x3FFFFA0`, mais le MSB est 1, donc le nombre est négatif, et nous pouvons le convertir manuellement en une forme pratique pour nous. D'après les règles de la négation, il faut simplement inverser tous les bits (ça donne `0b1011111=0x5F`), et ajouter 1 (`0x5F+1=0x60`). Donc le nombre

sous sa forme signée est -0x60. Multiplions -0x60 par 4 (car l'adresse est stockée dans l'opcode est divisée par 4) : ça fait -0x180. Maintenant calculons l'adresse de destination: 0x4005a0 + (-0x180) = 0x400420 (noter s'il vous plaît: nous considérons l'adresse de l'instruction BL, pas la valeur courante du PC, qui peut être différente!). Donc l'adresse de destination est 0x400420.

Plus d'informations relatives aux relogements en ARM64: [*ELF for the ARM 64-bit Architecture (AArch64)*, (2013)][191].

1.40 Détails spécifiques MIPS

1.40.1 Charger une constante 32-bit dans un registre

```
unsigned int f()
{
        return 0x12345678;
};
```

Toutes les instructions MIPS, tout comme en ARM, ont une taille de 32-bit, donc il n'est pas possible d'inclure une constante 32-bit dans une instruction.

Donc il faut utiliser au moins deux instructions: la première charge la partie haute du nombre de 32-bit et la seconde effectue une opération OR, qui met effectivement la partie 16-bit basse du registre de destination:

Listing 1.419 : GCC 4.4.5 -O3 (résultat en sortie de l'assembleur)

```
        li      $2,305397760  # 0x12340000
        j       $31
        ori     $2,$2,0x5678 ; slot de délai de branchement
```

IDA reconnaît ce pattern de code, qui se rencontre fréquemment, donc, par commodité, il montre la dernière instruction ORI comme la pseudo-instruction LI qui charge soit disant un nombre entier de 32-bit dans le registre $V0.

Listing 1.420 : GCC 4.4.5 -O3 (IDA)

```
         lui     $v0, 0x1234
         jr      $ra
         li      $v0, 0x12345678 ; slot de délai de branchement
```

La sortie de l'assembleur GCC a la pseudo instruction LI, mais il s'agit en fait ici de LUI («Load Upper Immediate» charger la valeur immédiate en partie haute), qui stocke une valeur 16-bit dans la partie haute du registre.

Regardons la sortie de *objdump* :

Listing 1.421 : objdump

```
00000000 <f>:
```

[191]Aussi disponible en http://infocenter.arm.com/help/topic/com.arm.doc.ihi0056b/IHI0056B_aaelf64.pdf

```
   0:   3c021234        lui     v0,0x1234
   4:   03e00008        jr      ra
   8:   34425678        ori     v0,v0,0x5678
```

Charger une variable globale 32-bit dans un registre

```
unsigned int global_var=0x12345678;

unsigned int f2()
{
        return global_var;
};
```

Ceci est légèrement différent: LUI charge les 16-bit haut de *global_var* dans $2 (ou $V0) et ensuite LW charge les 16-bit bas en l'ajoutant au contenu de $2:

Listing 1.422 : GCC 4.4.5 -O3 (résultat en sortie de l'assembleur)

```
f2:
        lui     $2,%hi(global_var)
        lw      $2,%lo(global_var)($2)
        j       $31
        nop     ; slot de délai de branchement

        ...

global_var:
        .word   305419896
```

IDA reconnaît cette paire d'instructions fréquemment utilisée, donc il concatène en une seule instruction LW.

Listing 1.423 : GCC 4.4.5 -O3 (IDA)

```
_f2:
                lw      $v0, global_var
                jr      $ra
                or      $at, $zero      ; slot de délai de branchement

                ...

                .data
                .globl global_var
global_var:     .word 0x12345678         # DATA XREF: _f2
```

La sortie d'*objdump* est la même que la sortie assembleur de GCC: Affichons le code de relogement du fichier objet:

Listing 1.424 : objdump

```
objdump -D filename.o

...
```

```
0000000c <f2>:
   c:   3c020000        lui     v0,0x0
  10:   8c420000        lw      v0,0(v0)
  14:   03e00008        jr      ra
  18:   00200825        move    at,at   ; slot de délai de branchement
  1c:   00200825        move    at,at

Désassemblage de la section .data:

00000000 <global_var>:
   0:   12345678        beq     s1,s4,159e4 <f2+0x159d8>

...

objdump -r filename.o

...

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
0000000c R_MIPS_HI16       global_var
00000010 R_MIPS_LO16       global_var

...
```

Nous voyons que l'adresse de *global_var* est écrite dans les instructions LUI et LW lors du chargement de l'exécutable: la partie haute de *global_var* se trouve dans la première (LUI), la partie basse dans la seconde (LW).

1.40.2 Autres lectures sur les MIPS

Dominic Sweetman, *See MIPS Run, Second Edition*, (2010).

Chapitre 2

Fondamentaux importants

2.1 Types intégraux

Un type intégral est un type de données dont les valeurs peuvent être converties en nombres. Les types intégraux comportent les nombres, les énumérations et les booléens.

2.1.1 Bit

Les valeurs booléennes sont une utilisation évidente des bits: 0 pour *faux* et 1 pour *vrai*.

Plusieurs valeurs booléennes peuvent être regroupées en un mot : Un mot de 32 bits contiendra 32 valeur booléennes, etc. On appelle *bitmap* ou *bitfield* un tel assemblage.

Cette approche engendre un surcoût de traitement: décalages, extraction, etc. A l'inverse l'utilisation d'un mot (ou d'un type *int*) pour chaque booléen gaspille de l'espace, au profit des performances.

Dans les environnements C/C++, la valeur 0 représente *faux* et toutes les autres valeurs *vrai*. Par exemple:

```
	if (1234)
		printf ("toujours exécuté\n");
	else
		printf ("jamais exécuté\n");
```

Une manière courante d'énumérer les caractères d'une chaîne en langage C:

```
	char *input=...;

	while(*input) // exécute le corps si le caractère *input est
    différent de zéro
	{
		// utiliser *input
```

```
                input++;
        };
```

2.1.2 Nibble

AKA demi-octet, tétrade. Représente 4 bits.

Toutes ces expressions sont toujours en usage.

Binary-Coded decimal (BCD[1])

Les demi-octets ont été utilisés par des CPU 4-bits tel que le Intel 4004 (utilisé dans les calculatrices).

On notera que la représentation *binary-coded decimal* (BCD) a été utilisée pour représenter les nombres sur 4 bits. L'entier 0 est représenté par la valeur 0b0000, l'entier 9 par 0b1001 tandis que les valeurs supérieures ne sont pas utilisées. La valeur décimale 1234 est ainsi représentée par 0x1234. Il est évident que cette représentation n'est pas la plus efficace en matière d'espace.

Elle possède en revanche un avantage: la conversion des nombres depuis et vers le format BCD est extrêmement simple. Les nombres au format BCD peuvent être additionnés, soustraits, etc., au prix d'une opération supplémentaire de gestion des demi-retenues. Les CPUs x86 proposent pour cela quelques instructions assez rares: AAA/DAA (gestion de la demi-retenue après addition), AAS/DAS (gestion de la demi-retenue après soustraction), AAM (après multiplication), AAD (après division).

Le support par les CPUs des nombres au format BCD est la raison d'être des *half-carry flag* (sur 8080/Z80) et *auxiliary flag* (AF sur x86). Ils représentent la retenue générée après traitement des 4 bits de poids faible (d'un octet). Le drapeau est utilisé par les instructions de gestion de retenue ci-dessus.

Le livre [Peter Abel, *IBM PC assembly language and programming* (1987)] doit sa popularité à la facilité de ces conversions. Hormis ce livre, l'auteur de ces notes n'a jamais rencontré en pratique de nombres au format BCD, sauf dans certains *nombres magiques* (5.6.1 on page 925), tels que lorsque la date de naissance d'un individu est encodé sous la forme 0x19791011—qui n'est autre qu'un nombre au format BCD.

Étonnement, j'ai trouvé que des nombres encodés BCD sont utilisés dans le logiciel SAP: https://yurichev.com/blog/SAP/. Certains nombres, prix inclus, sont encodés en format BCD dans la base de données. Peut-être ont-ils utilisé ce format pour être compatible avec d'anciens logiciels ou matériel?

Les instructions x86 destinées au traitement des nombres BCD ont parfois été utilisées à d'autres fins, le plus souvent non documentées, par exemple:

```
        cmp al,10
        sbb al,69h
        das
```

[1]Binary-Coded Decimal

Ce fragment de code abscons converties les nombres de 0 à 15 en caratères ASCII ’0’..’9’, ’A’..’F’.

Z80

Le processeur Z80 était un clone de la CPU 8 bits 8080 d’Intel. Par manque de place, il utilisait une UAL de 4 bits. Chaque opération impliquant deux nombres de 8 bits devait être traitée en deux étapes. Il en a découlé une utilisation naturelle des *half-carry flag*.

2.1.3 Caractère

A l’heure actuelle, l’utilisation de 8 bits par caractère est pratique courante. Il n’en a pas toujours été ainsi. Les cartes perforées utilisées pour les télétypes ne pouvaient comporter que 5 ou 6 perforations par caractères, et donc autant de bits.

Le terme *octet* met l’accent sur l’utilisation de 8 bits.: *fetchmail* est un de ceux qui utilise cette terminologie.

Sur les architectures à 36 bits, l’utilisation de 9 bits par caractère a été utilisée: un mot pouvait contenir 4 caractères. Ceci explique peut-être que le standard C/C++ indique que le type *char* doit supporter *au moins* 8 bits, mais que l’utilisation d’un nombre plus importants de bits est autorisé.

Par exemple, dans l’un des premiers ouvrage sur le langage C [2], nous trouvons :

```
char  one byte character (PDP-11, IBM360: 8 bits; H6070: 9 bits)
```

H6070 signifie probablement Honeywell 6070, qui comprenait des mots de 36 bits.

table ASCII standard

La représentation ASCII des caractères sur 7 bits constitue un standard, qui supporte donc 128 caractères différents. Les premiers logiciels de transport de mails fonctionnaient avec des codes ASCII sur 7 bits. Le standard MIME[3] nécessitait donc l’encodage des messages rédigés avec des alphabets non latins. Le code ASCII sur 7 bits a ensuite été augmenté d’un bit de parité qui a abouti à la représentation sur 8 bits.

Les clefs de chiffrage utilisées par *Data Encryption Standard* (DES[4]) comportent 56 bits, soit 8 groupes de 7 bits ce qui laisse un espace pour un bit de parité dans chaque groupe.

La mémorisation de la table ASCII est inutile. Il suffit de se souvenir de certains intervalles. [0..0x1F] sont les caractères de contrôle (non imprimables). [0x20..0x7E]

[2] https://yurichev.com/mirrors/C/bwk-tutor.html
[3] Multipurpose Internet Mail Extensions
[4] Data Encryption Standard

sont les caractères imprimables. Les codes à partir de la valeur 0x80 sont généralement utilisés pour les caractères non latins et pour certains caractères pseudo graphiques.

Quelques valeurs typiques à mémoriser sont : 0 (terminateur d'une chaîne de caractères en C, `'\0'` et C/C++) ; 0xA ou 10 (*fin de ligne*, `'\n'` en C/C++) ; 0xD ou 13 (*retour chariot*, `'\r'` en C/C++).

0x20 (espace).

CPUs 8 bits

Les processeurs x86 - descendants des CPUs 8080 8 bits - supportent la manipulation d'octet(s) au sein des registres. Les CPUs d'architecture RISC telles que les processeurs ARM et MIPS n'offrent pas cette possibilité.

2.1.4 Alphabet élargi

Il s'agit d'une tentative de supporter des langues non européennes en étendant le stockage d'un caractère à 16 bits. L'exemple le plus connu en est le noyau Windows NT et les fonctions win32 suffixées d'un *W*. Cet encodage est nommé UCS-2 ou UTF-16. Son utilisation explique la présence d'octets à zéro entre chaque caractère d'un texte en anglais ne comportant que des caractères latins.

En règle général, la notation *wchar_t* est un synonyme du type *short* qui utilise 16 bits.

2.1.5 Entier signé ou non signé

Certains s'étonneront qu'il existe un type de données entier non signé (positif ou nul) puisque chaque entier de ce type peut être représenté par un entier signé (positif ou négatif). Certes, mais le fait de ne pas avoir à utiliser un bit pour représenter le signe permet de doubler la taille de l'intervalle des valeurs qu'il est possible de représenter. Ainsi un octet signé permet de représenter les valeurs de -128 à +127, et l'octet non signé les valeurs de 0 à 255. Un autre avantage d'utiliser un type de données non signé est l'auto-documentation: vous définissez une variable qui ne peut pas recevoir de valeurs négatives.

L'absence de type de données non signées dans le langage Java a été critiqué. L'implémentation d'algorithmes cryptographiques à base d'opérations booléennes avec les seuls types de données signées est compliquée.

Une valeur telle que 0xFFFFFFFF (-1) est souvent utilisée, en particulier pour représenter un code d'erreur.

2.1.6 Mot

mot Le terme de 'mot' est quelque peu ambigu et dénote en général un type de données dont la taille correspond à celle d'un GPR. L'utilisation d'octets est pratique pour le stockage des caractères, mais souvent inadapté aux calculs arithmétiques.

C’est pourquoi, nombre de CPUs possèdent des GPRs dont la taille est de 16, 32 ou 64 bits. Les CPUs 8 bits tels que le 8080 et le Z80 proposent quant à eux de travailler sur des paires de registres 8 bits, dont chacune constitue un *pseudo-registre* de 16 bits. (*BC*, *DE*, *HL*, etc.). Les capacités des paires de registres du Z80 en font, en quelque sorte, un émulateur d’une CPU 16 bits.

En règle générale, un CPU présenté comme ”CPU n-bits” possède des GPRs dont la taille est de n bits.

À une certaine époque, les disques durs et les barettes de RAM étaient caractérisés comme ayant *n* kilo-mots et non pas *b* kilooctets/megaoctets.

Par exemple, *Apollo Guidance Computer* possède 2048 mots de RAM. S’agissant d’un ordinateur 16 bits, il y avait donc 4096 octets de RAM.

La mémoire magnétique du *TX-0* était de 64K mots de 18 bits, i.e., 64 kilo-mots.

DECSYSTEM-2060 pouvait supporter jusqu’à 4096 kilo mots de *solid state memory* (i.e., hard disks, tapes, etc). S’agissant d’un ordinateur 36 bits, cela représentait 18432 kilo octets ou 18 mega octets.

En fait, pourquoi auriez-vous besoin d’octets si vous avez des mots? Surtout pour le traitement des chaînes de texte. Les mots peuvent être utilisés dans presque toutes les autres situations.

---

int en C/C++ est presque systématiquement représenté par un mot. (L’architecture AMD64 fait exception car le type *int* possède une taille de 32 bits, peut-être pour une meilleure portabilité.)

Le type *int* est représenté sur 16 bits par le PDP-11 et les anciens compilateurs MS-DOS. Le type *int* est représenté sur 32 bits sur VAX, ainsi que sur l’architecture x86 à partir du 80386, etc.

De plus, dans les programmes C/C++, le type *int* est utilisé par défaut lorsque le type d’une variable n’est pas explicitement déclaré. Cette pratique peut apparaître comme un héritage du langage de programmation B[5].

L’accès le plus rapide à une variable s’effectue lorsqu’elle est contenue dans un GPR, plus même qu’un ensemble de bits, et parfois même plus rapide qu’un octet (puisqu’il n’est pas besoin d’isoler un bit ou un octet au sein d’un GPR). Ceci reste vrai même lorsque le registre est utilisé comme compteur d’itération d’une boucle de 0 à 99.

En langage assembleur x86, un mot représente 16 bits, car il en était ainsi sur les processeurs 8086 16 bits. Un *Double word* représente 32 bits, et un *quad word* 64 bits. C’est pourquoi, les mots de 16 bits sont déclarés par `DW` en assembleur x86, ceux de 32 bits par `DD` et ceux de 64 bits par `DQ`.

[5] `http://yurichev.com/blog/typeless/`

Dans les architectures ARM, MIPS, etc... un mot représente 32 bits, on parlera alors de *demi-mot* pour les types sur 16 bits. En conséquence, un *double word* sur une architecture RISC 32 bits est un type de données qui représente 64 bits.

GDB utilise la terminologie suivante : *demi-mot* pour 16 bits, mot pour 32 bits et *mot géant* pour 64 bits.

Les environnements C/C++ 16 bits sur PDP-11 et MS-DOS définissent le type *long* comme ayant une taille de 32 bits, ce qui serait sans doute une abréviation de *long word* ou de *long int*.

Les environnements C/C++ 32 bits définissent le type *long long* dont la taille est de 64 bits.

L'ambiguïté du terme *mot* est donc désormais évidente.

Dois-je utiliser le type *int* ?

Certains affirment que le type *int* ne doit jamais être utilisé, l'ambiguïté de sa définition pouvant être génératrice de bugs. A une certaine époque, la bibliothèque bien connue *lzhuf* utilisais le type *int* et fonctionnait parfaitement sur les architectures 16 bits. Portée sur une architecture pour laquelle le type *int* représentait 32 bits, elle pouvait alors crasher: `http://yurichev.com/blog/lzhuf/`.

Des types de données moins ambigus sont définis dans le fichier *stdint.h* : *uint8_t*, *uint16_t*, *uint32_t*, *uint64_t*, etc.

Donald E. Knuth fut l'un de ceux qui proposa[6] d'utiliser pour ces différents types des dénominations aux consonances distinctes: *octet/wyde/tetrabyte/octabyte*. Cette pratique est cependant moins courante que celle consistant à inclure directement dans le nom du type les termes *u* (*unsigned*) ainsi que le nombre de bits.

Ordinateurs à base de mots

En dépit de l'ambiguïté du terme mot, les ordinateurs modernes restent conçus sur ce concept: la RAM ainsi que tous les niveaux de mémoire cache demeurent organisés en mots et non pas en octets. La notion d'octet reste prépondérante en marketing.

Les accès aux adresses mémoire et cache alignées sur des frontières de mots est souvent plus performante que lorsque l'adresse n'est pas alignée.

Afin de rendre performante l'utilisation des structures de données, il convient toujours de de prendre en compte la longueur du mot du CPU sur lequel sera exécuté le programme lors de la définition des structures de données. Certains compilateurs - mais pas tous - prennent en charge cet alignement.

2.1.7 Registre d'adresse

Ceux qui ont fait leur premières armes sur les processeurs x86 32 et 64 bits, ou les processeurs RISC des années 90 tels que ARM, MIPS ou PowerPC prennent pour

[6] `http://www-cs-faculty.stanford.edu/~uno/news98.html`

acquis que la taille du bus d'adresse est la même que celle d'un GPR ou d'un mot. Cependant, cette règle n'est pas toujours respectée sur d'autres architectures.

Le processeur 8 bits Z80 peut adresser 2^{16} octets, en utilisant une paire de registres 8 bits ou certains registres spécialisés (*IX*, *IY*). En outre sur ce processeur les registres *SP* et *PC* contiennent 16 bits.

Le super calculateur Cray-1 possèdent des registres généraux de 64-bit, et des registres d'adressage de 24 bits. Il peut donc adresser 2^{24} octets, soit (16 mega mots ou 128 mega octets). La RAM coûtait très cher, et un Cray typique avait 1048576 (0x100000) mots de RAM, soit 8MB. Dans les années 70, la RAM était très coûteuse. Il paraissait alors inconcevable qu'un tel calculateur atteigne les 128 Mo. Dès lors pourquoi aurait-on utilisé des registres 64 bits pour l'adressage?

Les processeurs 8086/8088 utilisent un schéma d'adressage particulièrement bizarre: Les valeurs de deux registres de 16 bits sont additionnées de manière étrange afin d'obtenir une adresse sur 20 bits. S'agirait-il d'une sorte de virtualisation gadget (11.7 on page 1297)? Les processeurs 8086 pouvaient en effet faire fonctionner plusieurs programmes côte à côte (mais pas simultanément bien sûr).

Les premiers processeurs ARM1 implémentent un artefact intéressant:

> Un autre point intéressant est l'absence de quelques bits dans le registre PC. Le processeur ARM1 utilisant des adresses sur 26 bits, les 6 bits de poids fort ne sont pas utilisés. Comme toutes les adresses sont alignées sur une frontière de 32 bits, les deux bits les moins significatifs du registre PC sont toujours égaux à 0. Ces 8 bits sont non seulement inutilisés mais purement et simplement absents du processeur.

(`http://www.righto.com/2015/12/reverse-engineering-arm1-ancestor-of.html`)

En conséquence, il n'est pas possible d'affecter au registre PC une valeur dont l'un des deux bits de poids faible est différent de 0, pas plus qu'il n'est possible de positionner à 1 l'un des 6 bits de poids fort.

L'architecture x86-64 utilise des pointeurs et des adresses sur 64 bits, cependant en interne la largeur du bus d'adresse est de 48 bits, (ce qui est suffisant pour adresser 256 Tera octets de RAM).

2.1.8 Nombres

A quoi sont utilisés les nombres ?

Lorsque vous constatez que la valeur d'un registre de la CPU est modifié selon un certain motif, vous pouvez chercher à comprendre à quoi correspond ce motif. La capacité à déterminer le type de données qui découle de ce motif est une compétence précieuse pour le reverse engineer .

Booléen

Si le nombre alterne entre les valeurs 0 et 1, il y a des chances importantes pour qu'il s'agisse d'une valeur booléenne.

Compteur de boucle, index dans un tableau

Une variable dont la valeur augmente régulièrement en partant de 0, tel que 0, 1, 2, 3…— est probablement un compteur de boucle et/ou un index dans un tableau.

Nombres signés

Si vous constatez qu'une variable contient parfois des nombres très petits et d'autre fois des nombres très grands, tels que 0, 1, 2, 3, et 0xFFFFFFFF, 0xFFFFFFFE, 0xFFFFFFFD, il est probable qu'il s'agisse d'un entier signé sous forme de *two's complement* auquel cas les 3 dernières valeurs représentent en réalité -1, -2, -3.

Nombres sur 32 bits

Il existe des nombres tellement grands, qu'il existe une notation spéciale pour les représenter (Notation exponentielle de Knuth's) De tels nombres sont tellement grands qu'ils s'avèrent peu pratiques pour l'ingénierie, les sciences ou les mathématiques.

La plupart des ingénieurs et des scientifiques sont donc ravis d'utiliser la notation IEEE 754 pour les nombres flottants à double précision, laquelle peut représenter des valeurs allant jusqu'à $1.8 \cdot 10^{308}$. (En comparaison, le nombre d'atomes dans l'univers observable est estimé être entre $4 \cdot 10^{79}$ et $4 \cdot 10^{81}$.)

De fait, la limite supérieure des nombres utilisés dans les opérations concrètes est très très inférieure.

Pareil à l'époque de MS-DOS: les *int* 16 bits étaient utilisés pratiquement pour tout (indice de tableau, compteur de boucle), tandis que le type *long* sur 32 bits ne l'était que rarement.

Durant l'avènement de l'architecture x86-64, il fut décidé que le type *int* conserverait une taille de 32 bits, probablement parce que l'utilisation d'un type *int* de 64 bits est encore plus rare.

Je dirais que les nombre sur 16 bits qui couvrent l'intervalle 0..65535 sont probablement les nombres les plus utilisés en informatique.

Ceci étant, si vous rencontrez des nombres sur 32 bits particulièrement élevé tels que 0x87654321, il existe une bonne chance qu'il s'agisse :

- Il peut toujours s'agir d'un entier sur 16 bits, mais signé lorsque la valeur est entre 0xFFFF8000 (-32768) et 0xFFFFFFFF (-1).
- une adresse mémoire (ce qui peut être vérifié en utilisant les fonctionnalités de gestion mémoire du débogueur).
- des octets compactés (ce qui peut être vérifié visuellement).
- un ensemble de drapeaux binaires.

- de la cryptographie (amateur).
- un nombre magique (5.6.1 on page 925).
- un nombre flottant utilisant la représentation IEEE 754 (également vérifiable).

Il en va à peu près de même pour les valeurs sur 64 bits.

…donc un *int* sur 16 bits est suffisant pour à peu près n'importe quoi?

Il est intéressant de constater que dans [Michael Abrash, *Graphics Programming Black Book*, 1997 chapitre 13] nous pouvons lire qu'il existe pléthore de cas pour lesquels des variables sur 16 bits sont largement suffisantes. Dans le même temps, Michael Abrash se plaint que les CPUs 80386 et 80486 disposent de si peu de registres et propose donc de placer deux registres de 16 bits dans un registre de 32 bits et d'en effectuer des rotations en utilisant les instructions `ROR reg, 16` (sur 80386 et suivant) (`ROL reg, 16` fonctionne également) ou `BSWAP` (sur 80486 et suivant).

Cette approche rappelle celle du Z80 et de ses groupes de registres alternatifs (suffixés d'une apostrophe) vers lesquels le CPU pouvait être basculé (et inversement) au moyen de l'instruction EXX.

Taille des buffers

Lorssqu'un programmeur doit déclarer la taille d'un buffer, il utilise généralement une valeur de la forme 2^x (512 octets, 1024, etc.). Les valeurs de la forme 2^x sont faciles à reconnaître (1.28.5 on page 414) en décimal, en hexadécimal et en binaire.

Les programmeurs restent cependant des humains et conservent leur culture décimale. C'est pourquoi, dans le domaine des DBMS[7], la taille des champs textuels est souvent choisie sous la forme 10^x, 100, 200 par exemple. Ils pensent simplement «Okay, 100 suffira, attendez, 200 ira mieux». Et bien sûr, ils ont raison.

La taille maximale du type *VARCHAR2* dans Oracle RDBMS est de 4000 caractères et non de 4096.

Il n'y a rien à redire à ceci, ce n'est qu'un exemple d'utilisation des nombres sous la forme d'un multiple d'une puissance de dix.

Addresse

Garder à l'esprit une cartographie approximative de l'occupation mémoire du processus que vous déboguez est toujours une bonne idée. Ainsi, beaucoup d'exécutables win32 démarrent à l'adresse 0x00401000, donc une adresse telle que 0x00451230 se situe probablement dans sa section exécutable. Vous trouverez des adresses de cette sorte dans le registre EIP.

La pile est généralement située à une adresse inférieure à

[7] Database Management Systems

Beaucoup de débogueurs sont capables d'afficher la cartographie d'occupation mémoire du processus débogué, par exemple: 1.12.3 on page 109.

Une adresse qui augmente par pas de 4 sur une architecture 32-bit, ou par pas de 8 sur une architecture 64-bit constitue probablement l'énumération des adresses des éléments d'un tableau.

Il convient de savoir que win32 n'utilise pas les adresses inférieures à 0x10000, donc si vous observez un nombre inférieur à cette valeur, ce ne peut être une adresse (voir aussi `https://msdn.microsoft.com/en-us/library/ms810627.aspx`).

De toute manière, beaucoup de débogueurs savent vous indiquer si la valeur contenue dans un registre peut représenter l'adresse d'un élément. OllyDbg peut également vous afficher le contenu d'une chaîne de caractères ASCII si la valeur du registre est l'adresse d'une telle chaîne.

Drapeaux

Si vous observez une valeur pour laquelle un ou plusieurs bits changent de valeur de temps en temps tel que 0xABCD1234 → 0xABCD1434 et retour, il s'agit probablement d'un ensemble de drapeaux ou bitmap.

Compactage de caractères

Quand *strcmp()* ou *memcmp()* copient un buffer, ils traitent 4 (ou 8) octets à la fois. Donc, si une chaîne de caractères «4321» est recopié à une autre adresse, il adviendra un moment où vous observerez la valeur 0x31323334 dans un registre. Il s'agit du bloc de 4 caractères traité comme un entier sur 32 bits.

2.1.9 AND/OR/XOR au lieu de MOV

`OR reg, 0xFFFFFFFF` mets tous les bits à 1, en conséquence, peu importe ce qui se trouvait avant dans le registre, il sera mis à -1. `OR reg, -1` est plus court que `MOV reg, -1`, donc MSVC utilise OR au lieu de ce dernier suivant, par exemple: 3.18.1 on page 677.

De même, `AND reg, 0` efface tous les bits, par conséquent, elle se comporte comme `MOV reg, 0`.

`XOR reg, reg`, peu importe ce qui se trouvait précédemment dans le registre, efface tous les bits et se comporte donc comme `MOV reg, 0`.

2.2 Endianness

L'endianness (boutisme) est la façon de représenter les valeurs en mémoire.

2.2.1 Big-endian

La valeur `0x12345678` est représentée en mémoire comme:

adresse en mémoire	valeur de l’octet
+0	0x12
+1	0x34
+2	0x56
+3	0x78

Les CPUs big-endian comprennent les Motorola 68k, IBM POWER.

2.2.2 Little-endian

La valeur `0x12345678` est représentée en mémoire comme:

adresse en mémoire	valeur de l’octet
+0	0x78
+1	0x56
+2	0x34
+3	0x12

Les CPUs little-endian comprennent les Intel x86. Un exemple important d’utilisation de little-endian dans ce livre est: 1.35 on page 522.

2.2.3 Exemple

Prenons une système Linux MIPS big-endian déjà installé et prêt dans QEMU [8].

Et compilons cet exemple simple:

```
#include <stdio.h>

int main()
{
        int v;

        v=123;

        printf ("%02X %02X %02X %02X\n",
                *(char*)&v,
                *(((char*)&v)+1),
                *(((char*)&v)+2),
                *(((char*)&v)+3));
};
```

Après l’avoir lancé nous obtenons:

```
root@debian-mips:~# ./a.out
00 00 00 7B
```

C’est ça. 0x7B est 123 en décimal. En architecture little-endian, 7B est le premier octet (vous pouvez vérifier en x86 ou en x86-64, mais ici c’est le dernier, car l’octet le plus significatif vient en premier.

[8]Disponible au téléchargement ici: https://people.debian.org/~aurel32/qemu/mips/

C'est pourquoi il y a des distributions Liux séparées pour MIPS («mips» (big-endian) et «mipsel» (little-endian)). Il est impossible pour un binaire compilé pour une architecture de fonctionner sur un OS avec une architecture différente.

Il y a un exemple de MIPS big-endian dans ce livre: 1.30.4 on page 471.

2.2.4 Bi-endian

Les CPUs qui peuvent changer d'endianness sont les ARM, PowerPC, SPARC, MIPS, IA64[9], etc.

2.2.5 Convertir des données

L'instruction BSWAP peut être utilisée pour la conversion.

Les paquets de données des réseaux TCP/IP utilisent la convention bit-endian, c'est donc pourquoi un programme travaillant en architecture little-endian doit convertir les valeurs. Les fonctions `htonl()` et `htons()` sont utilisées en général.

En TCP/IP, big-endian est aussi appelé «network byte order», tandis que l'ordre des octets sur l'ordinateur «host byte order». Le «host byte order» est little-endian sur les x86 Intel et les autres architectures little-endian, mais est big-endian sur les IBM POWER, donc `htonl()` et `htons()` ne modifient aucun octet sur cette dernière.

2.3 Mémoire

Il y a 3 grands types de mémoire:

- Mémoire globale AKA «allocation statique de mémoire». Pas besoin de l'allouer explicitement, l'allocation est effectuée juste en déclarant des variables/-tableaux globalement. Ce sont des variables globales, se trouvant dans le segment de données ou de constantes. Elles sont accessibles globalement (ce qui est considéré comme un anti-pattern). Ce n'est pas pratique pour les buffers/-tableaux, car ils doivent avoir une taille fixée. Les débordements de tampons se produisant ici le sont en général en récrivant les variables ou les buffers se trouvant à côté d'eux en mémoire. Il y a un exemple dans ce livre: 1.12.3 on page 105.

- Stack (pile) AKA «allocation sur la pile». L'allocation est effectuée simplement en déclarant des variables/ tableaux localement dans la fonction. Ce sont en général des variables locales de la fonction. Parfois ces variables locales sont aussi visibles depuis les fonctions appelées, si l'appelant passe un pointeur sur une variable à la fonction appelée qui va être exécutée). L'allocation et la désallocation sont très rapide, il suffit de décaler SP.

  Mais elles ne conviennent pas non plus pour les tampons/tableaux, car la taille du tampon doit être fixée, à moins qu'`alloca()` (1.9.2 on page 49) (ou un tableau de longueur variable) ne soit utilisé. Les débordements de tampons écrasent en général les structures de pile importantes: 1.26.2 on page 350.

[9]Intel Architecture 64 (Itanium)

- Heap (tas) AKA «allocation dynamique de mémoire». L'allocation/dé-allocation est effectuée en appelant `malloc()/free()` ou `new/delete` en C++. Ceci est la méthode la plus pratique: la taille du bloc peut être définie lors de l'exécution.

  Il est possible de redimensionner (en utilisant `realloc()`), mais ça peut être long. Ceci est le moyen le plus lent d'allouer de la mémoire: L'allocation de mémoire doit gérer et mettre à jour toutes les structures de contrôle pendant l'allocation et la dé-allocation. Les débordements de tampons écrasent en général ces structures. L'allocation sur le tas est aussi la source des problème de fuite de mémoire: chaque bloc de mémoire doit être dé-alloué explicitement, mais on peut oublier de le faire, ou le faire de manière incorrecte.

  Un autre problème est l'«utilisation après la libération»—-utiliser un bloc de mémoire après que `free()` ait été appelé, ce qui est très dangereux.

  Exemple dans ce livre: 1.30.2 on page 447.

2.4 CPU

2.4.1 Prédicteurs de branchement

Certains des derniers compilateurs essayent d'éliminer les instructions de saut. Il y a des exemples dans ce livre: 1.18.1 on page 178, 1.18.3 on page 188, 1.28.5 on page 424.

C'est parce que le prédicteur de branchement n'est pas toujours parfait, donc les compilateurs essayent de faire sans les sauts conditionnels, si possible.

Les instructions conditionnelles en ARM (comme ADRcc) sont une manière, une autre est l'instruction x86 CMOVcc.

2.4.2 Dépendances des données

Les CPUs modernes sont capables d'exécuter des instructions simultanément (OOE[10]), mais pour ce faire, le résultat d'une instruction dans un groupe ne doit pas influencer l'exécution des autres. Par conséquent, le compilateur s'efforce d'utiliser des instructions avec le minimum d'influence sur l'état du CPU.

C'est pourquoi l'instruction LEA est si populaire, car elle ne modifie pas les flags du CPU, tandis que d'autres instructions arithmétiques le font.

2.5 Fonctions de hachage

Un exemple très simple est CRC32, un algorithme qui fournit des checksum plus «fort» à des fins de vérifications d'intégrité. Il est impossible de restaurer le texte d'origine depuis la valeur du hash, il a beaucoup moins d'informations: Mais CRC32 n'est pas cryptographiquement sûr: on sait comment modifier un texte afin que son

[10]Out-of-Order Execution

hash CRC32 résultant soit celui que l'on veut. Les fonctions cryptographiques sont protégées contre cela.

MD5, SHA1, etc. sont de telles fonctions et elles sont largement utilisées pour hacher les mots de passe des utilisateurs afin de les stocker dans une base de données. En effet: la base de données d'un forum Internet ne doit pas contenir les mots de passe des utilisateurs (une base de données volée compromettrait tous les mots de passe des utilisateurs) mais seulement les hachages (donc un cracker ne pourrait pas révéler les mots de passe). En outre, un forum Internet n'a pas besoin de connaître votre mot de passe exactement, il a seulement besoin de vérifier si son hachage est le même que celui dans la base de données, et vous donne accès s'ils correspondent. Une des méthodes de cracking la plus simple est simplement d'essayer de hacher tous les mots de passe possible pour voir celui qui correspond à la valeur recherchée. D'autres méthodes sont beaucoup plus complexes.

2.5.1 Comment fonctionnent les fonctions à sens unique?

Une fonction à sens unique est une fonction qui est capable de transformer une valeur en une autre, tandis qu'il est impossible (ou très difficile) de l'inverser. Certaines personnes éprouvent des difficultés à comprendre comment ceci est possible. Voici une démonstration simple.

Nous avons un vecteur de 10 nombres dans l'intervalle 0..9, chacun est présent une seule fois, par exemple:

```
4 6 0 1 3 5 7 8 9 2
```

L'algorithme pour une fonction à sens unique la plus simple possible est:

- prendre le nombre à l'indice zéro (4 dans notre cas) ;
- prendre le nombre à l'indice 1 (6 dans notre cas) ;
- échanger les nombres aux positions 4 et 6.

Marquons les nombres aux positions 4 et 6:

```
4 6 0 1 3 5 7 8 9 2
        ^   ^
```

Échangeons-les et nous obtenons ce résultat:

```
4 6 0 1 7 5 3 8 9 2
```

En regardant le résultat, et même si nous connaissons l'algorithme, nous ne pouvons pas connaître l'état initial de façon certaine, car les deux premiers nombres pourraient être 0 et/ou 1, et pourraient donc participer à la procédure d'échange.

Ceci est un exemple extrêmement simplifié pour la démonstration. Les fonctions à sens unique réelles sont bien plus complexes.

Chapitre 3

Exemples un peu plus avancés

3.1 Registre à zéro

Il manque un registre à zéro dans l'architecture x86, contrairement à MIPS et ARM. Toutefois, il arrive souvent que lorsqu'un compilateur assigne zéro à un registre, il y reste jusqu'à la fin de la fonction.

C'est le cas dans le jeu Mahjong de Windows 7 x86. EBX mis à zéro (en 0x010281BF) est utilisé pour initialiser les variables locales, passer un argument à zéro aux autres fonctions et pour comparer des valeurs avec lui

Listing 3.1 : Mahjong.exe de Windows 7 x86

```
.text:010281AE sub_10281AE     proc near  ; CODE XREF: sub_1028790+4FFp
.text:010281AE                            ; sub_102909A+357p ...
.text:010281AE
.text:010281AE var_34          = dword ptr -34h
.text:010281AE var_30          = dword ptr -30h
.text:010281AE var_2C          = dword ptr -2Ch
.text:010281AE var_28          = dword ptr -28h
.text:010281AE var_24          = dword ptr -24h
.text:010281AE var_20          = dword ptr -20h
.text:010281AE var_1C          = dword ptr -1Ch
.text:010281AE var_18          = dword ptr -18h
.text:010281AE var_14          = dword ptr -14h
.text:010281AE var_10          = dword ptr -10h
.text:010281AE var_4           = dword ptr -4
.text:010281AE arg_0           = dword ptr  8
.text:010281AE arg_4           = byte ptr  0Ch
.text:010281AE
.text:010281AE                 push    28h
.text:010281B0                 mov     eax, offset __ehhandler ...
.text:010281B5                 call    __EH_prolog3
.text:010281BA                 mov     edi, ecx
```

```
.text:010281BC                 mov     esi, [ebp+arg_0]
.text:010281BF                 xor     ebx, ebx                ; *
.text:010281C1                 mov     [ebp+var_10], ebx       ; *
.text:010281C4                 cmp     [esi], ebx              ; *
.text:010281C6                 jbe     short loc_10281E8
.text:010281C8
.text:010281C8 loc_10281C8:                        ; CODE XREF: sub_10281AE+38j
.text:010281C8                 mov     eax, [esi+0Ch]
.text:010281CB                 mov     ecx, [ebp+var_10]
.text:010281CE                 push    dword ptr [eax+ecx*4]
.text:010281D1                 call    sub_10506C9
.text:010281D6                 mov     eax, [ebp+var_10]
.text:010281D9                 pop     ecx
.text:010281DA                 mov     ecx, [esi+0Ch]
.text:010281DD                 mov     [ecx+eax*4], ebx        ; *
.text:010281E0                 inc     eax
.text:010281E1                 mov     [ebp+var_10], eax
.text:010281E4                 cmp     eax, [esi]
.text:010281E6                 jb      short loc_10281C8
.text:010281E8
.text:010281E8 loc_10281E8:                        ; CODE XREF: sub_10281AE+18j
.text:010281E8                 mov     [esi], ebx              ; *
.text:010281EA                 mov     [edi+14h], ebx          ; *
.text:010281ED                 mov     [ebp+var_34], ebx       ; *
.text:010281F0                 mov     [ebp+var_30], ebx       ; *
.text:010281F3                 mov     [ebp+var_2C], 10h
.text:010281FA                 mov     [ebp+var_28], ebx       ; *
.text:010281FD                 mov     [ebp+var_4], ebx        ; *
.text:01028200                 mov     [ebp+arg_0], ebx        ; *
.text:01028203                 cmp     [edi+0B0h], ebx         ; *
.text:01028209                 jbe     loc_10282C3
.text:0102820F
.text:0102820F loc_102820F:                        ; CODE XREF: sub_10281AE+10Fj
.text:0102820F                 mov     eax, [edi+0BCh]
.text:01028215                 mov     ecx, [ebp+arg_0]
.text:01028218                 mov     eax, [eax+ecx*4]
.text:0102821B                 mov     [ebp+var_14], eax
.text:0102821E                 cmp     eax, ebx                ; *
.text:01028220                 jz      loc_10282A6
.text:01028226                 push    ebx                     ; *
.text:01028227                 push    eax
.text:01028228                 mov     ecx, edi
.text:0102822A                 call    sub_1026B3D
.text:0102822F                 test    al, al
.text:01028231                 jz      short loc_10282A6
.text:01028233                 mov     [ebp+var_24], ebx       ; *
.text:01028236                 mov     [ebp+var_20], ebx       ; *
.text:01028239                 mov     [ebp+var_1C], 10h
.text:01028240                 mov     [ebp+var_18], ebx       ; *
.text:01028243                 lea     eax, [ebp+var_34]
.text:01028246                 push    eax
.text:01028247                 lea     eax, [ebp+var_24]
.text:0102824A                 push    eax
```

```
.text:0102824B                 push    [ebp+var_14]
.text:0102824E                 mov     ecx, edi
.text:01028250                 mov     byte ptr [ebp+var_4], 1
.text:01028254                 call    sub_1026E4F
.text:01028259                 mov     [ebp+var_10], ebx        ; *
.text:0102825C                 cmp     [ebp+var_24], ebx        ; *
.text:0102825F                 jbe     short loc_102829B
.text:01028261
.text:01028261 loc_1028261:                ; CODE XREF: sub_10281AE+EBj
.text:01028261                 push    0Ch; Size
.text:01028263                 call    sub_102E741
.text:01028268                 pop     ecx
.text:01028269                 cmp     eax, ebx                 ; *
.text:0102826B                 jz      short loc_1028286
.text:0102826D                 mov     edx, [ebp+var_10]
.text:01028270                 mov     ecx, [ebp+var_18]
.text:01028273                 mov     ecx, [ecx+edx*4]
.text:01028276                 mov     edx, [ebp+var_14]
.text:01028279                 mov     edx, [edx+4]
.text:0102827C                 mov     [eax], edx
.text:0102827E                 mov     [eax+4], ecx
.text:01028281                 mov     [eax+8], ebx             ; *
.text:01028284                 jmp     short loc_1028288
.text:01028286 ;
---------------------------------------------------------------------------
.text:01028286
.text:01028286 loc_1028286:                ; CODE XREF: sub_10281AE+BDj
.text:01028286                 xor     eax, eax
.text:01028288
.text:01028288 loc_1028288:                ; CODE XREF: sub_10281AE+D6j
.text:01028288                 push    eax
.text:01028289                 mov     ecx, esi
.text:0102828B                 call    sub_104922B
.text:01028290                 inc     [ebp+var_10]
.text:01028293                 mov     eax, [ebp+var_10]
.text:01028296                 cmp     eax, [ebp+var_24]
.text:01028299                 jb      short loc_1028261
.text:0102829B
.text:0102829B loc_102829B:                ; CODE XREF: sub_10281AE+B1j
.text:0102829B                 lea     ecx, [ebp+var_24]
.text:0102829E                 mov     byte ptr [ebp+var_4], bl
.text:010282A1                 call    sub_10349DB
.text:010282A6
.text:010282A6 loc_10282A6:                ; CODE XREF: sub_10281AE+72j
.text:010282A6                             ; sub_10281AE+83j
.text:010282A6                 push    [ebp+arg_0]
.text:010282A9                 lea     ecx, [ebp+var_34]
.text:010282AC                 call    sub_104922B
.text:010282B1                 inc     [ebp+arg_0]
.text:010282B4                 mov     eax, [ebp+arg_0]
.text:010282B7                 cmp     eax, [edi+0B0h]
.text:010282BD                 jb      loc_102820F
.text:010282C3
.text:010282C3 loc_10282C3:                ; CODE XREF: sub_10281AE+5Bj
```

```
.text:010282C3                 cmp     [ebp+arg_4], bl
.text:010282C6                 jz      short loc_1028337
.text:010282C8                 mov     eax, dword_1088AD8
.text:010282CD                 mov     esi, ds:EnableMenuItem
.text:010282D3                 mov     edi, 40002
.text:010282D8                 cmp     [eax+8], ebx    ; *
.text:010282DB                 jnz     short loc_10282EC
.text:010282DD                 push    3       ; uEnable
.text:010282DF                 push    edi     ; uIDEnableItem
.text:010282E0                 push    hMenu   ; hMenu
.text:010282E6                 call    esi ; EnableMenuItem
.text:010282E8                 push    3
.text:010282EA                 jmp     short loc_10282F7
.text:010282EC ;
---------------------------------------------------------------------------
.text:010282EC
.text:010282EC loc_10282EC:                    ; CODE XREF: sub_10281AE+12Dj
.text:010282EC                 push    ebx     ; *
.text:010282ED                 push    edi     ; uIDEnableItem
.text:010282EE                 push    hMenu   ; hMenu
.text:010282F4                 call    esi ; EnableMenuItem
.text:010282F6                 push    ebx     ; *
.text:010282F7
.text:010282F7 loc_10282F7:                    ; CODE XREF: sub_10281AE+13Cj
.text:010282F7                 push    edi     ; uIDEnableItem
.text:010282F8                 push    hmenu   ; hMenu
.text:010282FE                 call    esi ; EnableMenuItem
.text:01028300                 mov     ecx, dword_1088AD8
.text:01028306                 call    sub_1020402
.text:0102830B                 mov     edi, 40001
.text:01028310                 test    al, al
.text:01028312                 jz      short loc_1028321
.text:01028314                 push    ebx     ; *
.text:01028315                 push    edi     ; uIDEnableItem
.text:01028316                 push    hMenu   ; hMenu
.text:0102831C                 call    esi ; EnableMenuItem
.text:0102831E                 push    ebx     ; *
.text:0102831F                 jmp     short loc_102832E
.text:01028321 ;
---------------------------------------------------------------------------
.text:01028321
.text:01028321 loc_1028321:                    ; CODE XREF: sub_10281AE+164j
.text:01028321                 push    3       ; uEnable
.text:01028323                 push    edi     ; uIDEnableItem
.text:01028324                 push    hMenu   ; hMenu
.text:0102832A                 call    esi ; EnableMenuItem
.text:0102832C                 push    3       ; uEnable
.text:0102832E
.text:0102832E loc_102832E:                    ; CODE XREF: sub_10281AE+171j
.text:0102832E                 push    edi     ; uIDEnableItem
.text:0102832F                 push    hmenu   ; hMenu
.text:01028335                 call    esi ; EnableMenuItem
.text:01028337
.text:01028337 loc_1028337:                    ; CODE XREF: sub_10281AE+118j
.text:01028337                 lea     ecx, [ebp+var_34]
```

```
.text:0102833A                 call    sub_10349DB
.text:0102833F                 call    __EH_epilog3
.text:01028344                 retn    8
.text:01028344 sub_10281AE     endp
```

À propos, IDA peut renommer un registre à l'intérieur d'une fonction (presser n) :

Listing 3.2 : Mahjong.exe de Windows 7 x86

```
.text:010281AE sub_10281AE     proc near

...

.text:010281AE zero = ebx

...

.text:010281BF                 xor     zero, zero              ; *
.text:010281C1                 mov     [ebp+var_10], zero      ; *
.text:010281C4                 cmp     [esi], zero             ; *
.text:010281C6                 jbe     short loc_10281E8

...
```

3.2 Double négation

Une façon répandue[1] de convertir des valeurs différentes de zéro en 1 (ou le booléen *true*) et la valeur zéro en 0 (ou le booléen *false*) est la déclaration *!!variable* :

```
int convert_to_bool(int a)
{
        return !!a;
};
```

GCC 5.4 x86 avec optimisation:

```
convert_to_bool:
        mov     edx, DWORD PTR [esp+4]
        xor     eax, eax
        test    edx, edx
        setne   al
        ret
```

XOR efface toujours la valeur de retour dans EAX, même si SETNE n'est pas déclenché. I.e., XOR met la valeur de retour par défaut à zéro.

Si la valeur en entrée n'est pas égale à zéro (le suffixe -NE dans l'instruction SET), 1 est mis dans AL, autrement AL n'est pas modifié.

[1] C'est sujet à controverse, car ça conduit à du code difficile à lire

Pourquoi est-ce que SETNE opère sur la partie 8-bit basse du registre EAX ? Parce que ce qui compte c'est juste le dernier bit (0 or 1), puisque les autres bits sont mis à zéro par XOR.

Ainsi, ce code C/C++ peut être récrit comme ceci:

```
int convert_to_bool(int a)
{
        if (a!=0)
                return 1;
        else
                return 0;
};
```

...ou même:

```
int convert_to_bool(int a)
{
        if (a)
                return 1;
        else
                return 0;
};
```

Les compilateurs visant des CPUs n'ayant pas d'instructions similaires à SET, génèrent dans ce cas des instructions de branchement, etc.

3.3 const correctness

Ceci est une fonctionnalité indûment sous-utilisée de nombreux langages de programmation. Pour en savoir plus à ce sujet: 1, 2.

Idéalement, tout ce que vous ne modifiez pas devrait avoir le modificateur *const*. Il est intéressant de savoir comment le *const correctness* est implémenté à bas niveau. Il n'y a pas de vérification des variables ni des arguments de fonction *const* lors de l'exécution (seulement des vérifications lors de la compilation). Mais les variables globales de ce type sont allouées dans le segment de données en lecture seule.

Cet exemple va planter, car il est compilé par MSVC pour win32, la variable globale a est allouée dans le segment de données `.rdata` en lecture seule:

```
const a=123;

void f(int *i)
{
        *i=11; // crash
};

int main()
{
        f(&a);
        return a;
};
```

Les chaînes C *anonymes* (non liées à un nom de variable) ont aussi un type `const char*`. Vous ne pouvez pas les modifier:

```
#include <string.h>
#include <stdio.h>

void alter_string(char *s)
{
        strcpy (s, "Goodbye!");
        printf ("Result: %s\n", s);
};

int main()
{
        alter_string ("Hello, world!\n");
};
```

Ce code va planter sur Linux (“segmentation fault”) et sur Windows si il compilé par MinGW.

GCC pour Linux met toutes les chaînes de texte dans le segment de données `.rodata`, qui est explicitement en lecture seule (“read only data”) :

```
$ objdump -s 1

...

Contents of section .rodata:
 400600 01000200 52657375 6c743a20 25730a00  ....Result: %s..
 400610 48656c6c 6f2c2077 6f726c64 210a00    Hello, world!..
```

Lorsque la fonction `alter_string()` essaye d’y écrire, une exception se produit.

Les choses sont différentes dans le code généré par MSVC, les chaînes sont situées dans le segment `.data`, qui n’a pas de flag READONLY. Les développeurs de MSVC ont-ils fait un faux pas?

```
C:\...>objdump -s 1.exe

...

Contents of section .data:
 40b000 476f6f64 62796521 00000000 52657375  Goodbye!....Resu
 40b010 6c743a20 25730a00 48656c6c 6f2c2077  lt: %s..Hello, w
 40b020 6f726c64 210a0000 00000000 00000000  orld!...........
 40b030 01000000 00000000 c0cb4000 00000000  ..........@.....

...

C:\...>objdump -x 1.exe

...

Sections:
```

```
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00006d2a  00401000  00401000  00000400  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rdata        00002262  00408000  00408000  00007200  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00000e00  0040b000  0040b000  00009600  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .reloc        00000b98  0040e000  0040e000  0000a400  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

Toutefois, MinGW n'a pas cette erreur et alloue les chaînes de texte dans le segment `.rdata`.

3.3.1 Chaînes const se chevauchant

Le fait est qu'une chaîne C *anonyme* a un type *const* (1.5.1 on page 13), et que les chaînes C allouées dans le segment des constantes sont garanties d'être immuables, a cette conséquence intéressante: Le compilateur peut utiliser une partie spécifique de la chaîne.

Voyons cela avec un exemple:

```
#include <stdio.h>

int f1()
{
        printf ("world\n");
}

int f2()
{
        printf ("hello world\n");
}

int main()
{
        f1();
        f2();
}
```

La plupart des compilateurs C/C++ (MSVC inclus) allouent deux chaînes, mais voyons ce que fait GCC 4.8.1:

Listing 3.3 : GCC 4.8.1 + IDA

```
f1              proc near

s               = dword ptr -1Ch

                sub     esp, 1Ch
                mov     [esp+1Ch+s], offset s ; "world\n"
                call    _puts
                add     esp, 1Ch
```

```
                retn
f1              endp

f2              proc near

s               = dword ptr -1Ch

                sub     esp, 1Ch
                mov     [esp+1Ch+s], offset aHello ; "hello "
                call    _puts
                add     esp, 1Ch
                retn
f2              endp

aHello          db 'hello '
s               db 'world',0xa,0
```

Effectivement: lorsque nous affichons la chaîne «hello world» ses deux mots sont positionnés consécutivement en mémoire et l'appel à `puts()` depuis la fonction `f2()` n'est pas au courant que la chaîne est divisée. En fait, elle n'est pas divisée; elle l'est *virtuellement*, dans ce listing.

Lorsque `puts()` est appelé depuis `f1()`, il utilise la chaîne «world» ainsi qu'un octet à zéro. `puts()` ne sait pas qu'il y a quelque chose avant cette chaîne!

Cette astuce est souvent utilisée, au moins par GCC, et permet d'économiser de la mémoire. C'est proche du *string interning*.

Un autre exemple concernant ceci se trouve là: 3.4.

3.4 Exemple strstr()

Revenons au fait que GCC peut parfois utiliser une partie d'une chaîne de caractères: 3.3.1 on the preceding page.

La fonction *strstr()* de la bibliothèque standard C/C++ est utilisée pour trouver une occurrence dans une chaîne. C'est ce que nous voulons faire:

```
#include <string.h>
#include <stdio.h>

int main()
{
        char *s="Hello, world!";
        char *w=strstr(s, "world");

        printf ("%p, [%s]\n", s, s);
        printf ("%p, [%s]\n", w, w);
};
```

La sortie est:

```
0x8048530, [Hello, world!]
0x8048537, [world!]
```

La différence entre l'adresse de la chaîne originale et l'adresse de la sous-chaîne que *strstr()* a renvoyé est 7. En effet, la chaîne «Hello, » a une longueur de 7 caractères.

La fonction `printf()` lors du second appel n'a aucune idée qu'il y a des autres caractères avant la chaîne passée et elle affiche des caractères depuis le milieu de la chaîne originale jusqu'à la fin (marquée par un octet à zéro).

3.5 qsort() revisité

(Revenons au fait que l'instruction CMP fonctionne comme SUB : 1.12.4 on page 119.)

Maintenant que vous êtes déjà familier avec la fonction qsort() (1.33 on page 495), voici un bel exemple où l'opération de comparaison (CMP) peut être remplacée par l'opération de soustraction (SUB).

```
/* fonction de comparaison qsort int */
int int_cmp(const void *a, const void *b)
{
    const int *ia = (const int *)a; // casting de types de pointeur
    const int *ib = (const int *)b;
    return *ia  - *ib;
        /* comparaison d'entier: renvoie négatif si if b > a
        et positif si a > b */
}
```

(http://www.anyexample.com/programming/c/qsort__sorting_array_of_strings__integers_and_structs.xml http://archive.is/Hh3jz)

Aussi, une implémentation typique de `strcmp()` (tiré d'OpenBSD) :

```
int
strcmp(const char *s1, const char *s2)
{
        while (*s1 == *s2++)
                if (*s1++ == 0)
                        return (0);
        return (*(unsigned char *)s1 - *(unsigned char *)--s2);
}
```

3.6 Conversion de température

Un autre exemple très populaire dans les livres de programmation est un petit programme qui convertit une température de Fahrenheit vers Celsius ou inversement.

$$C = \frac{5 \cdot (F - 32)}{9}$$

Nous pouvons aussi ajouter une gestion des erreurs simples: 1) nous devons vérifier si l'utilisateur a entré un nombre correct; 2) nous devons tester si la température en Celsius n'est pas en dessous de -273 (qui est en dessous du zéro absolu, comme vu pendant les cours de physique à l'école)

La fonction exit() termine le programme instantanément, sans retourner à la fonction appelante.

3.6.1 Valeurs entières

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int celsius, fahr;
        printf ("Enter temperature in Fahrenheit:\n");
        if (scanf ("%d", &fahr)!=1)
        {
                printf ("Error while parsing your input\n");
                exit(0);
        };

        celsius = 5 * (fahr-32) / 9;

        if (celsius<-273)
        {
                printf ("Error: incorrect temperature!\n");
                exit(0);
        };
        printf ("Celsius: %d\n", celsius);
};
```

MSVC 2012 x86 avec optimisation

Listing 3.4 : MSVC 2012 x86 avec optimisation

```
$SG4228 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB      '%d', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius: %d', 0aH, 00H

_fahr$ = -4     ; taille = 4
_main   PROC
        push    ecx
        push    esi
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4228          ; 'Enter temperature in Fahrenheit:'
        call    esi                     ; appeler printf()
        lea     eax, DWORD PTR _fahr$[esp+12]
        push    eax
```

```
        push    OFFSET $SG4230          ; '%d'
        call    DWORD PTR __imp__scanf
        add     esp, 12
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4231          ; 'Error while parsing your input'
        call    esi                     ; appeler printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN9@main:
$LN2@main:
        mov     eax, DWORD PTR _fahr$[esp+8]
        add     eax, -32        ; ffffffe0H
        lea     ecx, DWORD PTR [eax+eax*4]
        mov     eax, 954437177  ; 38e38e39H
        imul    ecx
        sar     edx, 1
        mov     eax, edx
        shr     eax, 31         ; 0000001fH
        add     eax, edx
        cmp     eax, -273       ; fffffeefH
        jge     SHORT $LN1@main
        push    OFFSET $SG4233          ; 'Error: incorrect temperature!'
        call    esi     ; appeler printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN10@main:
$LN1@main:
        push    eax
        push    OFFSET $SG4234          ; 'Celsius: %d'
        call    esi     ; appeler printf()
        add     esp, 8
        ; renvoyer 0 - d'après le standard C99
        xor     eax, eax
        pop     esi
        pop     ecx
        ret     0
$LN8@main:
_main   ENDP
```

Ce que l'on peut en dire:

- L'adresse de `printf()` est d'abord chargée dans le registre ESI, donc les futurs appels à `printf()` seront faits juste par l'instruction CALL ESI. C'est une technique très populaire des compilateurs, possible si plusieurs appels consécutifs vers la même fonction sont présents dans le code, et/ou s'il y a un registre disponible qui peut être utilisé pour ça.
- Nous voyons l'instruction ADD EAX, -32 à l'endroit où 32 doit être soustrait de la valeur. $EAX = EAX + (-32)$ est équivalent à $EAX = EAX - 32$ et curieusement, le compilateur a décidé d'utiliser ADD au lieu de SUB. Peut-être que ça en vaut

la peine, difficile d'en être sûr.

- L'instruction LEA est utilisée quand la valeur est multipliée par 5: `lea ecx, DWORD PTR [eax+eax*4]`. Oui, $i + i * 4$ équivaut à $i * 5$ et LEA s'exécute plus rapidement que IMUL.
  D'ailleurs, la paire d'instructions `SHL EAX, 2 / ADD EAX, EAX` peut aussi être utilisée ici— certains compilateurs le font.

- La division par l'astuce de la multiplication (3.12 on page 636) est aussi utilisée ici.

- `main()` retourne 0 si nous n'avons pas `return 0` à la fin. Le standard C99 nous dit [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)5.1.2.2.3] que `main()` va retourner 0 dans le cas où la déclaration `return` est manquante. Cette règle fonctionne uniquement pour la fonction `main()`.

  Cependant, MSVC ne supporte pas officiellement C99, mais peut-être qu'il le supporte partiellement?

MSVC 2012 x64 avec optimisation

Le code est quasiment le même, mais nous trouvons une instruction `INT 3` après chaque appel à `exit()`.

```
        xor     ecx, ecx
        call    QWORD PTR __imp_exit
        int     3
```

`INT 3` est un point d'arrêt du debugger.

C'est connu que `exit()` est l'une des fonctions qui ne retourne jamais [2], donc si elle le fait, quelque chose de vraiment étrange est arrivé et il est temps de lancer le debugger.

3.6.2 Valeurs à virgule flottante

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        double celsius, fahr;
        printf ("Enter temperature in Fahrenheit:\n");
        if (scanf ("%lf", &fahr)!=1)
        {
                printf ("Error while parsing your input\n");
                exit(0);
        };

        celsius = 5 * (fahr-32) / 9;
```

[2] une autre connue est `longjmp()`

```
        if (celsius<-273)
        {
                printf ("Error: incorrect temperature!\n");
                exit(0);
        };
        printf ("Celsius: %lf\n", celsius);
};
```

MSVC 2010 x86 utilise des instructions FPU…

Listing 3.5 : MSVC 2010 x86 avec optimisation

```
$SG4038 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4040 DB      '%lf', 00H
$SG4041 DB      'Error while parsing your input', 0aH, 00H
$SG4043 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4044 DB      'Celsius: %lf', 0aH, 00H

__real@c071100000000000 DQ 0c071100000000000r   ; -273
__real@4022000000000000 DQ 04022000000000000r   ; 9
__real@4014000000000000 DQ 04014000000000000r   ; 5
__real@4040000000000000 DQ 04040000000000000r   ; 32

_fahr$ = -8     ; taille = 8
_main   PROC
        sub     esp, 8
        push    esi
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4038          ; 'Enter temperature in Fahrenheit:'
        call    esi                     ; appeler printf()
        lea     eax, DWORD PTR _fahr$[esp+16]
        push    eax
        push    OFFSET $SG4040          ; '%lf'
        call    DWORD PTR __imp__scanf
        add     esp, 12
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4041          ; 'Error while parsing your input'
        call    esi                     ; appeler printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN2@main:
        fld     QWORD PTR _fahr$[esp+12]
        fsub    QWORD PTR __real@4040000000000000 ; 32
        fmul    QWORD PTR __real@4014000000000000 ; 5
        fdiv    QWORD PTR __real@4022000000000000 ; 9
        fld     QWORD PTR __real@c071100000000000 ; -273
        fcomp   ST(1)
        fnstsw  ax
        test    ah, 65  ; 00000041H
        jne     SHORT $LN1@main
        push    OFFSET $SG4043          ; 'Error: incorrect temperature!'
        fstp    ST(0)
```

```
        call    esi             ; appeler printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN1@main:
        sub     esp, 8
        fstp    QWORD PTR [esp]
        push    OFFSET $SG4044          ; 'Celsius: %lf'
        call    esi
        add     esp, 12
        ; renvoyer 0 - d'après le standard C99
        xor     eax, eax
        pop     esi
        add     esp, 8
        ret     0
$LN10@main:
_main   ENDP
```

…mais MSVC 2012 utilise à la place des instructions SIMD :

Listing 3.6 : MSVC 2010 x86 avec optimisation

```
$SG4228 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB      '%lf', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius: %lf', 0aH, 00H
__real@c071100000000000 DQ 0c071100000000000r   ; -273
__real@4040000000000000 DQ 04040000000000000r   ; 32
__real@4022000000000000 DQ 04022000000000000r   ; 9
__real@4014000000000000 DQ 04014000000000000r   ; 5

_fahr$ = -8     ; taille = 8
_main   PROC
        sub     esp, 8
        push    esi
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4228          ; 'Enter temperature in Fahrenheit:'
        call    esi                     ; appeler printf()
        lea     eax, DWORD PTR _fahr$[esp+16]
        push    eax
        push    OFFSET $SG4230          ; '%lf'
        call    DWORD PTR __imp__scanf
        add     esp, 12
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4231          ; 'Error while parsing your input'
        call    esi     ; appeler printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN9@main:
$LN2@main:
        movsd   xmm1, QWORD PTR _fahr$[esp+12]
```

```
        subsd   xmm1, QWORD PTR __real@4040000000000000 ; 32
        movsd   xmm0, QWORD PTR __real@c071100000000000 ; -273
        mulsd   xmm1, QWORD PTR __real@4014000000000000 ; 5
        divsd   xmm1, QWORD PTR __real@4022000000000000 ; 9
        comisd  xmm0, xmm1
        jbe     SHORT $LN1@main
        push    OFFSET $SG4233          ; 'Error: incorrect temperature!'
        call    esi             ; appeler printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN10@main:
$LN1@main:
        sub     esp, 8
        movsd   QWORD PTR [esp], xmm1
        push    OFFSET $SG4234          ; 'Celsius: %lf'
        call    esi                     ; appeler printf()
        add     esp, 12
        ; renvoyer 0 - d'après le standard C99
        xor     eax, eax
        pop     esi
        add     esp, 8
        ret     0
$LN8@main:
_main   ENDP
```

Bien sûr, les instructions SIMD sont disponibles dans le mode x86, incluant celles qui fonctionnent avec les nombres à virgule flottante.

C'est un peu plus simple de les utiliser pour les calculs, donc le nouveau compilateur de Microsoft les utilise.

Nous pouvons aussi voir que la valeur -273 est chargée dans le registre XMM0 trop tôt. Et c'est OK, parce que le compilateur peut mettre des instructions dans un ordre différent de celui du code source.

3.7 Suite de Fibonacci

Un autre exemple très utilisé dans les livres de programmation est la fonction récursive qui génère les termes de la suite de Fibonacci[3]. Cette suite est très simple: chaque nombre consécutif est la somme des deux précédents. Les deux premiers termes sont 0 et 1 ou 1 et 1.

La suite commence comme ceci:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181...$$

3.7.1 Exemple #1

L'implémentation est simple. Ce programme génère la suite jusqu'à 21.

[3] http://oeis.org/A000045

```
#include <stdio.h>

void fib (int a, int b, int limit)
{
        printf ("%d\n", a+b);
        if (a+b > limit)
                return;
        fib (b, a+b, limit);
};

int main()
{
        printf ("0\n1\n1\n");
        fib (1, 1, 20);
};
```

Listing 3.7 : MSVC 2010 x86

```
_a$ = 8                 ; size = 4
_b$ = 12                ; size = 4
_limit$ = 16            ; size = 4
_fib    PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
        push    eax
        push    OFFSET $SG2643
        call    DWORD PTR __imp__printf
        add     esp, 8
        mov     ecx, DWORD PTR _a$[ebp]
        add     ecx, DWORD PTR _b$[ebp]
        cmp     ecx, DWORD PTR _limit$[ebp]
        jle     SHORT $LN1@fib
        jmp     SHORT $LN2@fib
$LN1@fib:
        mov     edx, DWORD PTR _limit$[ebp]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        call    _fib
        add     esp, 12
$LN2@fib:
        pop     ebp
        ret     0
_fib    ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
```

```
        push    OFFSET $SG2647 ; "0\n1\n1\n"
        call    DWORD PTR __imp__printf
        add     esp, 4
        push    20
        push    1
        push    1
        call    _fib
        add     esp, 12
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Nous allons illustrer les frames de pile avec ceci.

Chargeons cet exemple dans OllyDbg et traçons jusqu'au dernier appel de f() :

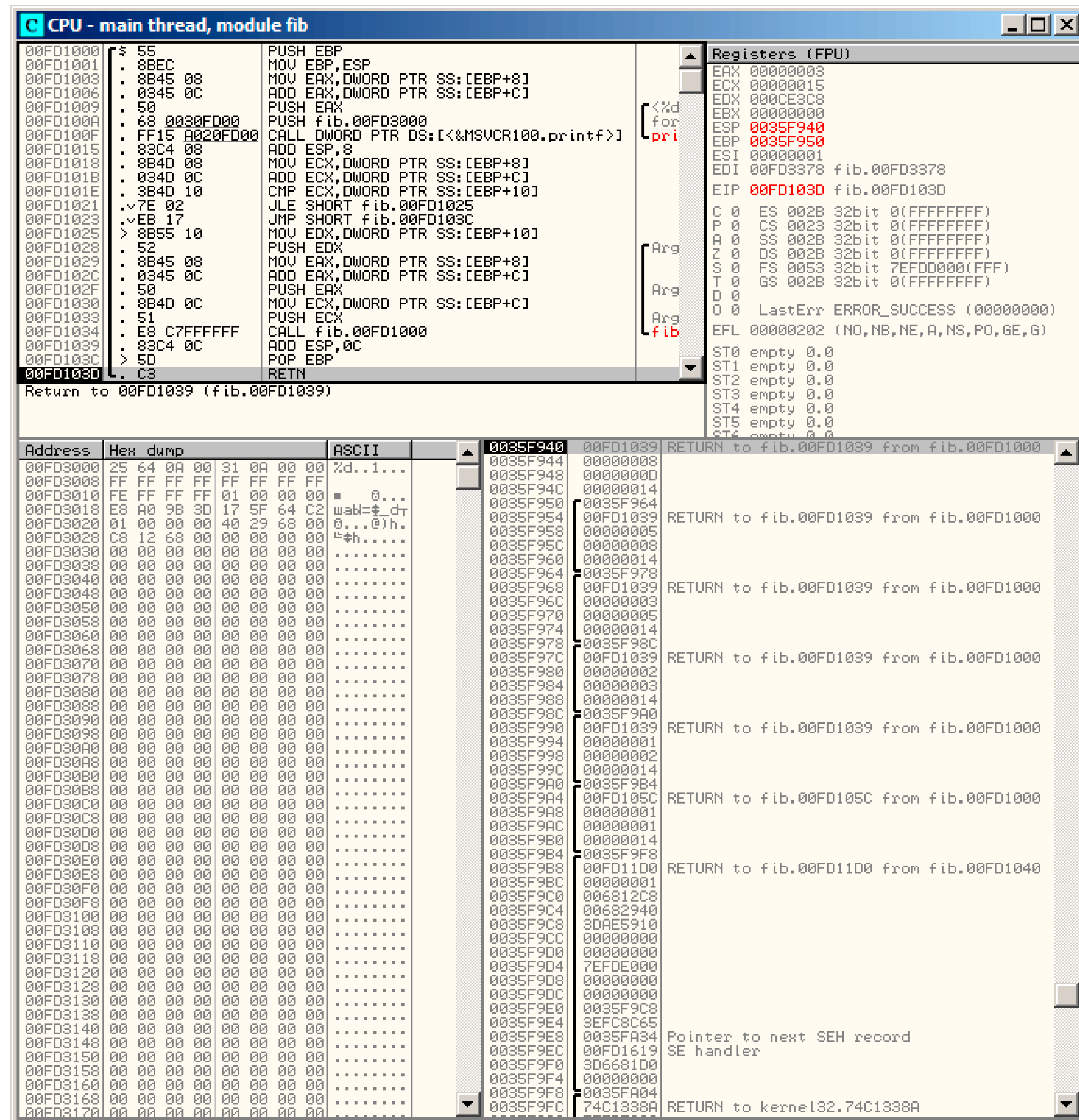


Fig. 3.1: OllyDbg : dernier appel de f()

Examinons plus attentivement la pile. Les commentaires ont été ajoutés par l'auteur de ce livre [4] :

```
0035F940   00FD1039  RETURN to fib.00FD1039 from fib.00FD1000
0035F944   00000008  1er  argument: a
0035F948   0000000D  2nd  argument b
0035F94C   00000014  3ème argument: limit
0035F950  /0035F964  registre EBP sauvé
0035F954  |00FD1039  RETURN to fib.00FD1039 from fib.00FD1000
0035F958  |00000005  1er  argument: a
0035F95C  |00000008  2nd  argument: b
0035F960  |00000014  3ème argument: limit
0035F964  ]0035F978  registre EBP sauvé
0035F968  |00FD1039  RETURN to fib.00FD1039 from fib.00FD1000
0035F96C  |00000003  1er  argument: a
0035F970  |00000005  2nd  argument: b
0035F974  |00000014  3ème argument: limit
0035F978  ]0035F98C  registre EBP sauvé
0035F97C  |00FD1039  RETURN to fib.00FD1039 from fib.00FD1000
0035F980  |00000002  1er  argument: a
0035F984  |00000003  2nd  argument: b
0035F988  |00000014  3ème argument: limit
0035F98C  ]0035F9A0  registre EBP sauvé
0035F990  |00FD1039  RETURN to fib.00FD1039 from fib.00FD1000
0035F994  |00000001  1er  argument: a
0035F998  |00000002  2nd  argument: b
0035F99C  |00000014  3ème argument: limit
0035F9A0  ]0035F9B4  registre EBP sauvé
0035F9A4  |00FD105C  RETURN to fib.00FD105C from fib.00FD1000
0035F9A8  |00000001  1er  argument: a     \
0035F9AC  |00000001  2nd  argument: b     | préparé dans main() pour f1()
0035F9B0  |00000014  3ème argument: limit /
0035F9B4  ]0035F9F8  registre EBP sauvé
0035F9B8  |00FD11D0  RETURN to fib.00FD11D0 from fib.00FD1040
0035F9BC  |00000001  main() 1er  argument: argc \
0035F9C0  |006812C8  main() 2nd  argument: argv | préparé dans CRT pour ↙
    ↪ main()
0035F9C4  |00682940  main() 3ème argument: envp /
```

La fonction est récursive [5], donc la pile ressemble à un «sandwich».

Nous voyons que l'argument *limit* est toujours le même (0x14 ou 20), mais que les arguments a et b sont différents pour chaque appel.

Il y a aussi ici le RA-s et la valeur sauvée de EBP. OllyDbg est capable de déterminer les frames basés sur EBP, donc il les dessine ces accolades. Les valeurs dans chaque accolade constituent la frame de pile, autrement dit, la zone de la pile qu'un appel de fonction utilise comme espace dédié.

Nous pouvons aussi dire que chaque appel de fonction ne doit pas accéder les éléments de la pile au delà des limites de son bloc (en excluant les arguments de la

[4]À propos, il est possible de sélectionner plusieurs entrées dans OllyDbg et de les copier dans le presse-papier (Crtl-C). C'est ce qui a été fait par l'auteur pour cet exemple.

[5]i.e., elle s'appelle elle-même

fonction), bien que cela soit techniquement possible.

C'est généralement vrai, à moins que la fonction n'ait des bugs.

Chaque valeur sauvée de EBP est l'adresse de la structure de pile locale précédente: c'est la raison pour laquelle certains débogueurs peuvent facilement diviser la pile en blocs et afficher chaque argument de la fonction.

Comme nous le voyons ici, chaque exécution de fonction prépare les arguments pour l'appel de fonction suivant.

À la fin, nous voyons les 3 arguments de `main()`. `argc` vaut 1 (oui, en effet, nous avons lancé le programme sans argument sur la ligne de commande).

Ceci peut conduire facilement à un débordement de pile: il suffit de supprimer (ou commenter) le test de la limite et ça va planter avec l'exception `0xC00000FD` (stack overflow).

3.7.2 Exemple #2

Ma fonction a quelques redondances, donc ajoutons une nouvelle variable locale *next* et remplaçons tout les «a+b» avec elle:

```
#include <stdio.h>

void fib (int a, int b, int limit)
{
        int next=a+b;
        printf ("%d\n", next);
        if (next > limit)
                return;
        fib (b, next, limit);
};

int main()
{
        printf ("0\n1\n1\n");
        fib (1, 1, 20);
};
```

C'est la sortie de MSVC sans optimisation, donc la variable *next* est allouée sur la pile locale:

Listing 3.8 : MSVC 2010 x86

```
_next$ = -4     ; size = 4
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_limit$ = 16    ; size = 4
_fib    PROC
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
```

```
        mov     DWORD PTR _next$[ebp], eax
        mov     ecx, DWORD PTR _next$[ebp]
        push    ecx
        push    OFFSET $SG2751 ; '%d'
        call    DWORD PTR __imp__printf
        add     esp, 8
        mov     edx, DWORD PTR _next$[ebp]
        cmp     edx, DWORD PTR _limit$[ebp]
        jle     SHORT $LN1@fib
        jmp     SHORT $LN2@fib
$LN1@fib:
        mov     eax, DWORD PTR _limit$[ebp]
        push    eax
        mov     ecx, DWORD PTR _next$[ebp]
        push    ecx
        mov     edx, DWORD PTR _b$[ebp]
        push    edx
        call    _fib
        add     esp, 12
$LN2@fib:
        mov     esp, ebp
        pop     ebp
        ret     0
_fib    ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2753 ; "0\n1\n1\n"
        call    DWORD PTR __imp__printf
        add     esp, 4
        push    20
        push    1
        push    1
        call    _fib
        add     esp, 12
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Chargeons-le à nouveau dans OllyDbg :

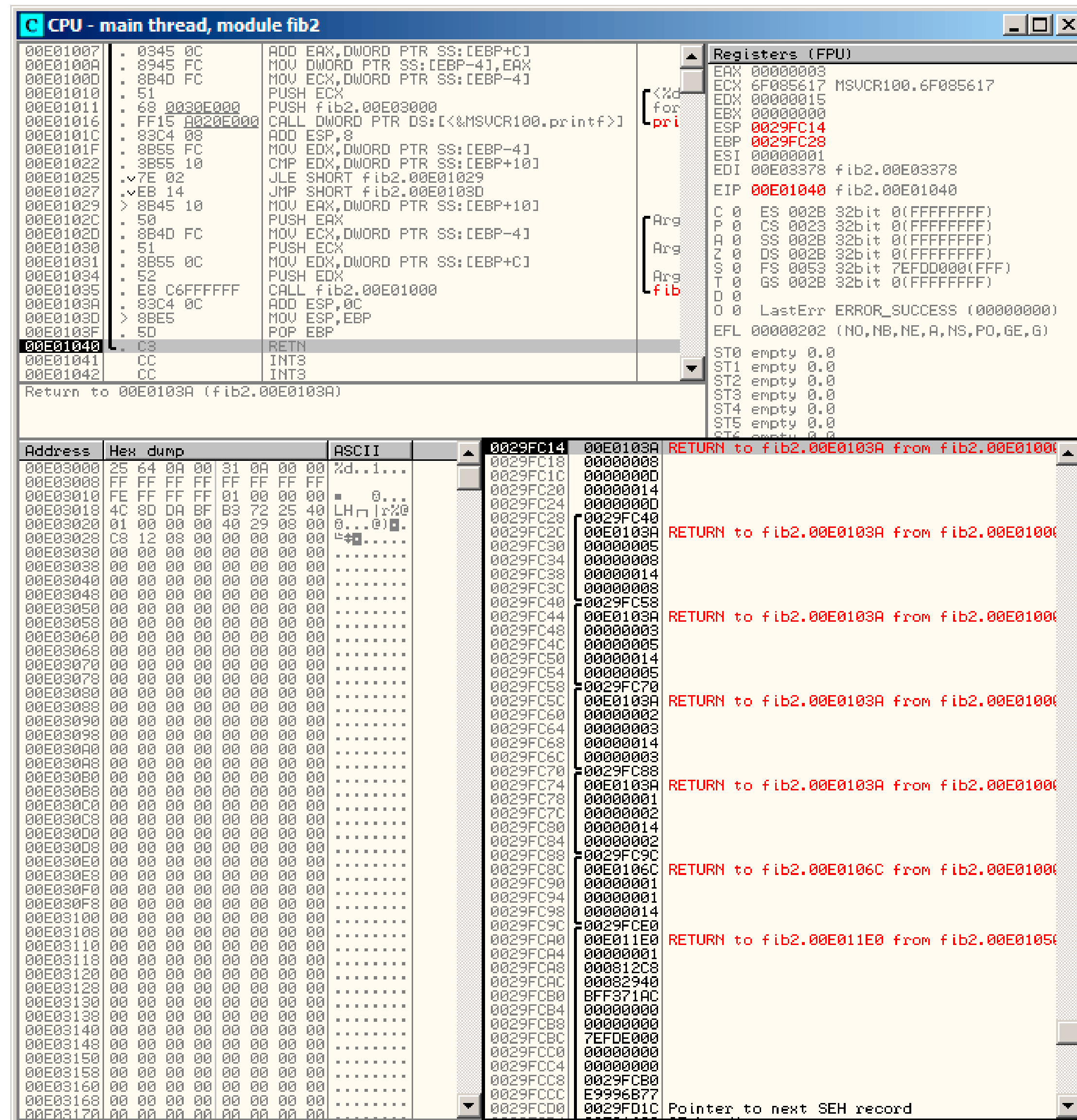


Fig. 3.2: OllyDbg : dernier appel de f()

Maintenant, la variable *next* est présente dans chaque frame.

Examinons plus attentivement la pile. L'auteur a de nouveau ajouté ses commentaires:

```
0029FC14   00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC18   00000008  1er  argument: a
0029FC1C   0000000D  2nd  argument: b
0029FC20   00000014  3ème argument: limit
0029FC24   0000000D  variable "next"
0029FC28  /0029FC40  registre EBP sauvé
0029FC2C  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC30  |00000005  1er  argument: a
0029FC34  |00000008  2nd  argument: b
0029FC38  |00000014  3ème argument: limit
0029FC3C  |00000008  "next" variable
0029FC40  ]0029FC58  registre EBP sauvé
0029FC44  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC48  |00000003  1er  argument: a
0029FC4C  |00000005  2nd  argument: b
0029FC50  |00000014  3ème argument: limit
0029FC54  |00000005  variable "next"
0029FC58  ]0029FC70  registre EBP sauvé
0029FC5C  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC60  |00000002  1er  argument: a
0029FC64  |00000003  2nd  argument: b
0029FC68  |00000014  3ème argument: limit
0029FC6C  |00000003  variable "next"
0029FC70  ]0029FC88  registre EBP sauvé
0029FC74  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC78  |00000001  1er  argument: a     \
0029FC7C  |00000002  2nd  argument: b     | préparé dans f1() pour le ↙
    ↳ prochain appel à f1()
0029FC80  |00000014  3ème argument: limit /
0029FC84  |00000002  variable "next"
0029FC88  ]0029FC9C  registre EBP sauvé
0029FC8C  |00E0106C  RETURN to fib2.00E0106C from fib2.00E01000
0029FC90  |00000001  1er  argument: a     \
0029FC94  |00000001  2nd  argument: b     | préparé dans main() pour f1()
0029FC98  |00000014  3ème argument: limit /
0029FC9C  ]0029FCE0  registre EBP sauvé
0029FCA0  |00E011E0  RETURN to fib2.00E011E0 from fib2.00E01050
0029FCA4  |00000001  main() 1er  argument: argc \
0029FCA8  |000812C8  main() 2nd  argument: argv | préparé dans CRT pour ↙
    ↳ main()
0029FCAC  |00082940  main() 3ème argument: envp /
```

Voici ce que l'on voit: la valeur *next* est calculée dans chaque appel de la fonction, puis passée comme argument b au prochain appel.

3.7.3 Résumé

Les fonctions récursives sont esthétiquement jolies, mais techniquement elles peuvent dégrader les performances à cause de leur usage intensif de la pile. Quiconque qui

écrit du code dont la perfomance est critique devrait probablement éviter la récursion.

Par exemple, j'ai écrit une fois une fonction pour chercher un nœud particulier dans un arbre binaire. Bien que la fonction récursive avait l'air élégante, il y avait du temps passé à chaque appel de fonction pour le prologue et l'épilogue, elle fonctionnait deux ou trois fois plus lentement que l'implémentation itérative (sans récursion).

À propos, c'est la raison pour laquelle certains compilateurs fonctionnels LP[6] (où la récursion est très utilisée) utilisent les appels terminaux. Nous parlons d'appel terminal lorsqu'une fonction a un seul appel à elle-même, situé à sa fin, comme:

Listing 3.9 : Scheme, exemple copié/collé depuis Wikipédia

```
;; factorial : nombre -> nombre
;; pour calculer le produit de tous les entiers
;; positifs inférieurs ou égaux à n.
(define (factorial n)
 (if (= n 1)
     1
         (* n (factorial (- n 1)))))
```

Les appels terminaux sont importants car le compilateur peut retravailler facilement ce code en un code itératif, pour supprimer la récursion.

3.8 Exemple de calcul de CRC32

C'est une technique très répandue de calcul de hachage basée sur une table CRC32[7].

```
/* By Bob Jenkins, (c) 2006, Public Domain */

#include <stdio.h>
#include <stddef.h>
#include <string.h>

typedef  unsigned long  ub4;
typedef  unsigned char  ub1;

static const ub4 crctab[256] = {
  0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419,
  0x706af48f, 0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4,
  0xe0d5e91e, 0x97d2d988, 0x09b64c2b, 0x7eb17cbd, 0xe7b82d07,
  0x90bf1d91, 0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de,
  0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7, 0x136c9856,
  0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9,
  0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4,
  0xa2677172, 0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
  0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940, 0x32d86ce3,
  0x45df5c75, 0xdcd60dcf, 0xabd13d59, 0x26d930ac, 0x51de003a,
  0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423, 0xcfba9599,
```

[6] LISP, Python, Lua, etc.
[7] Le code source provient d'ici: http://burtleburtle.net/bob/c/crc.c

```
  0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
  0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190,
  0x01db7106, 0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f,
  0x9fbfe4a5, 0xe8b8d433, 0x7807c9a2, 0x0f00f934, 0x9609a88e,
  0xe10e9818, 0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
  0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e, 0x6c0695ed,
  0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
  0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3,
  0xfbd44c65, 0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2,
  0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a,
  0x346ed9fc, 0xad678846, 0xda60b8d0, 0x44042d73, 0x33031de5,
  0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa, 0xbe0b1010,
  0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
  0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17,
  0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6,
  0x03b6e20c, 0x74b1d29a, 0xead54739, 0x9dd277af, 0x04db2615,
  0x73dc1683, 0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8,
  0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1, 0xf00f9344,
  0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
  0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a,
  0x67dd4acc, 0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5,
  0xd6d6a3e8, 0xa1d1937e, 0x38d8c2c4, 0x4fdff252, 0xd1bb67f1,
  0xa6bc5767, 0x3fb506dd, 0x48b2364b, 0xd80d2bda, 0xaf0a1b4c,
  0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55, 0x316e8eef,
  0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
  0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe,
  0xb2bd0b28, 0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31,
  0x2cd99e8b, 0x5bdeae1d, 0x9b64c2b0, 0xec63f226, 0x756aa39c,
  0x026d930a, 0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
  0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38, 0x92d28e9b,
  0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242,
  0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1,
  0x18b74777, 0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c,
  0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45, 0xa00ae278,
  0xd70dd2ee, 0x4e048354, 0x3903b3c2, 0xa7672661, 0xd06016f7,
  0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc, 0x40df0b66,
  0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
  0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605,
  0xcdd70693, 0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8,
  0x5d681b02, 0x2a6f2b94, 0xb40bbe37, 0xc30c8ea1, 0x5a05df1b,
  0x2d02ef8d
};

/* how to derive the values in crctab[] from polynomial 0xedb88320 */
void build_table()
{
  ub4 i, j;
  for (i=0; i<256; ++i) {
    j = i;
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
```

```
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    printf("0x%.8lx, ", j);
    if (i%6 == 5) printf("\n");
  }
}


/* the hash function */
ub4 crc(const void *key, ub4 len, ub4 hash)
{
  ub4  i;
  const ub1 *k = key;
  for (hash=len, i=0; i<len; ++i)
    hash = (hash >> 8) ^ crctab[(hash & 0xff) ^ k[i]];
  return hash;
}

/* To use, try "gcc -O crc.c -o crc; crc < crc.c" */
int main()
{
  char s[1000];
  while (gets(s)) printf("%.8lx\n", crc(s, strlen(s), 0));
  return 0;
}
```

Nous sommes seulement intéressés par la fonction `crc()`. À propos, faîtes attention aux deux déclarations d'initialisation dans la boucle `for()` : `hash=len, i=0`. Le standard C/C++ permet ceci, bien sûr. Le code généré contiendra deux opérations dans la partie d'initialisation de la boucle, au lieu d'une.

Compilons-le dans MSVC avec l'optimisation (/Ox). Dans un soucis de concision, seule la fonction `crc()` est listée ici, avec mes commentaires.

```
_key$ = 8                ; size = 4
_len$ = 12               ; size = 4
_hash$ = 16              ; size = 4
_crc    PROC
    mov    edx, DWORD PTR _len$[esp-4]
    xor    ecx, ecx ; i est stocké dans ECX
    mov    eax, edx
    test   edx, edx
    jbe    SHORT $LN1@crc
    push   ebx
    push   esi
    mov    esi, DWORD PTR _key$[esp+4] ; ESI = key
    push   edi
$LL3@crc:

; fonctionne avec des octets en utilisant seulement des registres 32-bit.
; l'octet à l'adresse key+i est stocké dans EDI
```

```
    movzx  edi, BYTE PTR [ecx+esi]
    mov    ebx, eax ; EBX = (hash = len)
    and    ebx, 255 ; EBX = hash & 0xff

; XOR EDI, EBX (EDI=EDI^EBX)
; cette opération utilise tous les 32 bits de chaque registre
; mais les autres bits (8-31) sont toujours mis à 0, donc c'est OK
; ils sont mis à 0 car, comme pour EDI, cela a été fait avec l'instruction
    MOVZX ci-dessus
; les bits hauts de EBX sont mis à 0 par l'instruction AND EBX, 255 ci-dessus
    (255 = 0xff)

    xor    edi, ebx

; EAX=EAX>>8; bits 24-31 pris de nul part seront mis à 0
    shr    eax, 8

; EAX=EAX^crctab[EDI*4] - choisir le EDI-ème élément de la table crctab[]
    xor    eax, DWORD PTR _crctab[edi*4]
    inc    ecx            ; i++
    cmp    ecx, edx       ; i<len ?
    jb     SHORT $LL3@crc ; oui
    pop    edi
    pop    esi
    pop    ebx
$LN1@crc:
    ret    0
_crc    ENDP
```

Essayons la même chose dans GCC 4.4.1 avec l'option -O3 :

```
                public crc
crc             proc near

key             = dword ptr  8
hash            = dword ptr  0Ch

                push    ebp
                xor     edx, edx
                mov     ebp, esp
                push    esi
                mov     esi, [ebp+key]
                push    ebx
                mov     ebx, [ebp+hash]
                test    ebx, ebx
                mov     eax, ebx
                jz      short loc_80484D3
                nop                   ; remplissage
                lea     esi, [esi+0]  ; remplissage; fonctionne comme NOP
                                      ; (ESI ne change pas ici)

loc_80484B8:
                mov     ecx, eax      ; sauve l'état pérécédent du hash dans
```

```
    ECX
                xor     al, [esi+edx]; AL=*(key+i)
                add     edx, 1       ; i++
                shr     ecx, 8       ; ECX=hash>>8
                movzx   eax, al      ; EAX=*(key+i)
                mov     eax, dword ptr ds:crctab[eax*4] ; EAX=crctab[EAX]
                xor     eax, ecx     ; hash=EAX^ECX
                cmp     ebx, edx
                ja      short loc_80484B8

loc_80484D3:
                pop     ebx
                pop     esi
                pop     ebp
                retn
crc             endp
```

GCC a aligné le début de la boucle sur une limite de 8-octet en ajoutant `NOP` et `lea esi, [esi+0]` (qui est aussi une opération sans effet).

Vous pouvez en lire plus à ce sujet dans la section npad (.1.7 on page 1347).

3.9 Exemple de calcul d'adresse réseau

Comme nous le savons, une adresse (IPv4) consiste en quatre nombres dans l'intervalle $0 \dots 255$, i.e., quatre octets.

Quatre octets peuvent être stockés facilement dans une variable 32-bit, donc une adresse IPv4 d'hôte, de masque réseau ou d'adresse de réseau peuvent toutes être un entier 32-bit.

Du point de vue de l'utilisateur, le masque réseau est défini par quatre nombres et est formaté comme 255.255.255.0, mais les ingénieurs réseaux (sysadmins) utilisent une notation plus compacte comme «/8 », «/16 », etc.

Cette notation défini simplement le nombre de bits qu'a le masque, en commençant par le MSB.

Masque	Hôte	Utilisable	Masque de réseau	Masque hexadécimal	
/30	4	2	255.255.255.252	0xfffffffc	
/29	8	6	255.255.255.248	0xfffffff8	
/28	16	14	255.255.255.240	0xfffffff0	
/27	32	30	255.255.255.224	0xffffffe0	
/26	64	62	255.255.255.192	0xffffffc0	
/24	256	254	255.255.255.0	0xffffff00	réseau de classe C
/23	512	510	255.255.254.0	0xfffffe00	
/22	1024	1022	255.255.252.0	0xfffffc00	
/21	2048	2046	255.255.248.0	0xfffff800	
/20	4096	4094	255.255.240.0	0xfffff000	
/19	8192	8190	255.255.224.0	0xffffe000	
/18	16384	16382	255.255.192.0	0xffffc000	
/17	32768	32766	255.255.128.0	0xffff8000	
/16	65536	65534	255.255.0.0	0xffff0000	réseau de classe B
/8	16777216	16777214	255.0.0.0	0xff000000	réseau de classe A

Voici un petit exemple, qui calcule l'adresse du réseau en appliquant le masque réseau à l'adresse de l'hôte.

```
#include <stdio.h>
#include <stdint.h>

uint32_t form_IP (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4)
{
        return (ip1<<24) | (ip2<<16) | (ip3<<8) | ip4;
};

void print_as_IP (uint32_t a)
{
        printf ("%d.%d.%d.%d\n",
                (a>>24)&0xFF,
                (a>>16)&0xFF,
                (a>>8)&0xFF,
                (a)&0xFF);
};

// bit=31..0
uint32_t set_bit (uint32_t input, int bit)
{
        return input=input|(1<<bit);
};

uint32_t form_netmask (uint8_t netmask_bits)
{
        uint32_t netmask=0;
        uint8_t i;

        for (i=0; i<netmask_bits; i++)
                netmask=set_bit(netmask, 31-i);

        return netmask;
};
```

```
void calc_network_address (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4, uint8_t netmask_bits)
{
        uint32_t netmask=form_netmask(netmask_bits);
        uint32_t ip=form_IP(ip1, ip2, ip3, ip4);
        uint32_t netw_adr;

        printf ("netmask=");
        print_as_IP (netmask);

        netw_adr=ip&netmask;

        printf ("network address=");
        print_as_IP (netw_adr);
};

int main()
{
        calc_network_address (10, 1, 2, 4, 24);    // 10.1.2.4, /24
        calc_network_address (10, 1, 2, 4, 8);     // 10.1.2.4, /8
        calc_network_address (10, 1, 2, 4, 25);    // 10.1.2.4, /25
        calc_network_address (10, 1, 2, 64, 26);   // 10.1.2.4, /26
};
```

3.9.1 calc_network_address()

La fonction `calc_network_address()` est la plus simple: elle effectue simplement un AND entre l'adresse de l'hôte et le masque de réseau, dont le résultat est l'adresse du réseau.

Listing 3.10 : MSVC 2012 avec optimisation /Ob0

```
_ip1$ = 8               ; size = 1
_ip2$ = 12              ; size = 1
_ip3$ = 16              ; size = 1
_ip4$ = 20              ; size = 1
_netmask_bits$ = 24     ; size = 1
_calc_network_address PROC
        push    edi
        push    DWORD PTR _netmask_bits$[esp]
        call    _form_netmask
        push    OFFSET $SG3045 ; 'netmask='
        mov     edi, eax
        call    DWORD PTR __imp__printf
        push    edi
        call    _print_as_IP
        push    OFFSET $SG3046 ; 'network address='
        call    DWORD PTR __imp__printf
        push    DWORD PTR _ip4$[esp+16]
        push    DWORD PTR _ip3$[esp+20]
        push    DWORD PTR _ip2$[esp+24]
        push    DWORD PTR _ip1$[esp+28]
        call    _form_IP
```

```
22          and     eax, edi        ; network address = host address & netmask
23          push    eax
24          call    _print_as_IP
25          add     esp, 36
26          pop     edi
27          ret     0
28  _calc_network_address ENDP
```

À la ligne 22, nous voyons le plus important AND—ici l'adresse du réseau est calculée.

3.9.2 form_IP()

La fonction `form_IP()` met juste les 4 octets dans une valeur 32-bit.

Voici comment cela est fait habituellement:

- Allouer une variable pour la valeur de retour. La mettre à 0.
- Prendre le 4ème octet (de poids le plus faible), appliquer l'opération OR à cet octet et renvoyer la valeur.
- Prendre le troisième octet, le décaler à gauche de 8 bits. Vous obtenez une valeur comme `0x0000bb00` où bb est votre troisième octet. Appliquer l'opération OR à la valeur résultante. La valeur de retour contenait `0x000000aa` jusqu'à présent, donc effectuer un OU logique des valeurs produira une valeur comme `0x0000bbaa`.
- Prendre le second octet, le décaler à gauche de 16 bits. Vous obtenez une valeur comme `0x00cc0000` où `cc` est votre deuxième octet. Appliquer l'opération OR à la valeur résultante. La valeur de retour contenait `0x0000bbaa` jusqu'à présent, donc effectuer un OU logique des valeurs produira une valeur comme `0x00ccbbaa`.
- Prendre le premier octet, le décaler à gauche de 24 bits. Vous obtenez une valeur comme `0xdd000000` où dd est votre premier octet. Appliquer l'opération OR à la valeur résultante. La valeur de retour contenait `0x00ccbbaa` jusqu'à présent, donc effectuer un OU logique des valeurs produira une valeur comme `0xddccbbaa`.

Voici comment c'est fait par MSVC 2012 sans optimisation:

Listing 3.11 : MSVC 2012 sans optimisation

```
; désigner ip1 comme "dd", ip2 comme "cc", ip3 comme "bb", ip4 comme "aa".
_ip1$ = 8       ; taille = 1
_ip2$ = 12      ; taille = 1
_ip3$ = 16      ; taille = 1
_ip4$ = 20      ; taille = 1
_form_IP PROC
        push    ebp
        mov     ebp, esp
        movzx   eax, BYTE PTR _ip1$[ebp]
        ; EAX=000000dd
        shl     eax, 24
```

```
        ; EAX=dd000000
        movzx   ecx, BYTE PTR _ip2$[ebp]
        ; ECX=000000cc
        shl     ecx, 16
        ; ECX=00cc0000
        or      eax, ecx
        ; EAX=ddcc0000
        movzx   edx, BYTE PTR _ip3$[ebp]
        ; EDX=000000bb
        shl     edx, 8
        ; EDX=0000bb00
        or      eax, edx
        ; EAX=ddccbb00
        movzx   ecx, BYTE PTR _ip4$[ebp]
        ; ECX=000000aa
        or      eax, ecx
        ; EAX=ddccbbaa
        pop     ebp
        ret     0
_form_IP ENDP
```

Certes, l'ordre est différent, mais, bien sûr, l'ordre des opérations n'a pas d'importance.

MSVC 2012 avec optimisation produit en fait la même chose, mais d'une façon différente:

Listing 3.12 : MSVC 2012 avec optimisation /Ob0

```
; désigner ip1 comme "dd", ip2 comme "cc", ip3 comme "bb", ip4 comme "aa".
_ip1$ = 8       ; taille = 1
_ip2$ = 12      ; taille = 1
_ip3$ = 16      ; taille = 1
_ip4$ = 20      ; taille = 1
_form_IP PROC
        movzx   eax, BYTE PTR _ip1$[esp-4]
        ; EAX=000000dd
        movzx   ecx, BYTE PTR _ip2$[esp-4]
        ; ECX=000000cc
        shl     eax, 8
        ; EAX=0000dd00
        or      eax, ecx
        ; EAX=0000ddcc
        movzx   ecx, BYTE PTR _ip3$[esp-4]
        ; ECX=000000bb
        shl     eax, 8
        ; EAX=00ddcc00
        or      eax, ecx
        ; EAX=00ddccbb
        movzx   ecx, BYTE PTR _ip4$[esp-4]
        ; ECX=000000aa
        shl     eax, 8
        ; EAX=ddccbb00
        or      eax, ecx
```

```
        ; EAX=ddccbbaa
        ret     0
_form_IP ENDP
```

Nous pourrions dire que chaque octet est écrit dans les 8 bits inférieurs de la valeur de retour, et qu'elle est ensuite décalée à gauche d'un octet à chaque étape.

Répéter 4 fois pour chaque octet en entrée.

C'est tout! Malheureusement, il n'y sans doute pas d'autre moyen de le faite.

Il n'y a pas de CPUs ou d'ISAs répandues qui possède une instruction pour composer une valeur à partir de bits ou d'octets.

C'est d'habitude fait par décalage de bit et OU logique.

3.9.3 print_as_IP()

La fonction `print_as_IP()` effectue l'inverse: séparer une valeur 32-bit en 4 octets.

Le découpage fonctionne un peu plus simplement: il suffit de décaler la valeur en entrée de 24, 16, 8 ou 0 bits, prendre les 8 bits d'indice 0 à 7 (octet de poids faible), et c'est fait:

Listing 3.13 : MSVC 2012 sans optimisation

```
_a$ = 8                 ; size = 4
_print_as_IP PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        ; EAX=ddccbbaa
        and     eax, 255
        ; EAX=000000aa
        push    eax
        mov     ecx, DWORD PTR _a$[ebp]
        ; ECX=ddccbbaa
        shr     ecx, 8
        ; ECX=00ddccbb
        and     ecx, 255
        ; ECX=000000bb
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        ; EDX=ddccbbaa
        shr     edx, 16
        ; EDX=0000ddcc
        and     edx, 255
        ; EDX=000000cc
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        ; EAX=ddccbbaa
        shr     eax, 24
        ; EAX=000000dd
        and     eax, 255 ; sans doute une instruction redondante
        ; EAX=000000dd
```

```
        push    eax
        push    OFFSET $SG2973 ; '%d.%d.%d.%d'
        call    DWORD PTR __imp__printf
        add     esp, 20
        pop     ebp
        ret     0
_print_as_IP ENDP
```

MSVC 2012 avec optimisation fait presque la même chose, mais sans recharger inutilement la valeur en entrée:

Listing 3.14 : MSVC 2012 avec optimisation /Ob0

```
_a$ = 8         ; size = 4
_print_as_IP PROC
        mov     ecx, DWORD PTR _a$[esp-4]
        ; ECX=ddccbbaa
        movzx   eax, cl
        ; EAX=000000aa
        push    eax
        mov     eax, ecx
        ; EAX=ddccbbaa
        shr     eax, 8
        ; EAX=00ddccbb
        and     eax, 255
        ; EAX=000000bb
        push    eax
        mov     eax, ecx
        ; EAX=ddccbbaa
        shr     eax, 16
        ; EAX=0000ddcc
        and     eax, 255
        ; EAX=000000cc
        push    eax
        ; ECX=ddccbbaa
        shr     ecx, 24
        ; ECX=000000dd
        push    ecx
        push    OFFSET $SG3020 ; '%d.%d.%d.%d'
        call    DWORD PTR __imp__printf
        add     esp, 20
        ret     0
_print_as_IP ENDP
```

3.9.4 form_netmask() et set_bit()

La fonction `form_netmask()` produit un masque de réseau à partir de la notation CIDR[8]. Bien sûr, il serait plus efficace d'utiliser une sorte de table pré-calculée, mais nous utilisons cette façon de faire intentionnellement, afin d'illustrer les décalages de bit.

[8]Classless Inter-Domain Routing

Nous allons aussi écrire une fonction séparées `set_bit()`. Ce n'est pas une très bonne idée de créer un fonction pour une telle opération primitive, mais cela facilite la compréhension du fonctionnement.

Listing 3.15 : MSVC 2012 avec optimisation /Ob0

```
_input$ = 8             ; size = 4
_bit$ = 12              ; size = 4
_set_bit PROC
        mov     ecx, DWORD PTR _bit$[esp-4]
        mov     eax, 1
        shl     eax, cl
        or      eax, DWORD PTR _input$[esp-4]
        ret     0
_set_bit ENDP

_netmask_bits$ = 8      ; size = 1
_form_netmask PROC
        push    ebx
        push    esi
        movzx   esi, BYTE PTR _netmask_bits$[esp+4]
        xor     ecx, ecx
        xor     bl, bl
        test    esi, esi
        jle     SHORT $LN9@form_netma
        xor     edx, edx
$LL3@form_netma:
        mov     eax, 31
        sub     eax, edx
        push    eax
        push    ecx
        call    _set_bit
        inc     bl
        movzx   edx, bl
        add     esp, 8
        mov     ecx, eax
        cmp     edx, esi
        jl      SHORT $LL3@form_netma
$LN9@form_netma:
        pop     esi
        mov     eax, ecx
        pop     ebx
        ret     0
_form_netmask ENDP
```

`set_bit()` est primitive: elle décale juste 1 à gauche du nombre de bits dont nous avons besoin et puis effectue un OU logique avec la valeur «input». `form_netmask()` a une boucle: elle met autant de bits (en partant du MSB) que demandé dans l'argument `netmask_bits`.

3.9.5 Résumé

C'est tout! Nous le lançons et obtenons:

```
netmask=255.255.255.0
network address=10.1.2.0
netmask=255.0.0.0
network address=10.0.0.0
netmask=255.255.255.128
network address=10.1.2.0
netmask=255.255.255.192
network address=10.1.2.64
```

3.10 Boucles: quelques itérateurs

Dans la plupart des cas, les boucles ont un seul itérateur, mais elles peuvent en avoir plusieurs dans le code résultant.

Voici un exemple très simple:

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
        size_t i;

        // copier d'un tableau a l'autre selon un schema bizarre
        for (i=0; i<cnt; i++)
                a1[i*3]=a2[i*7];
};
```

Il y a deux multiplications à chaque itération et ce sont des opérations coûteuses. Est-ce que ça peut être optimisé d'une certaine façon?

Oui, si l'on remarque que les deux indices de tableau prennent des valeurs qui peuvent être facilement calculées sans multiplication.

3.10.1 Trois itérateurs

Listing 3.16 : MSVC 2013 x64 avec optimisation

```
f       PROC
; RCX=a1
; RDX=a2
; R8=cnt
        test    r8, r8          ; cnt==0? sortir si oui
        je      SHORT $LN1@f
        npad    11
$LL3@f:
        mov     eax, DWORD PTR [rdx]
        lea     rcx, QWORD PTR [rcx+12]
        lea     rdx, QWORD PTR [rdx+28]
        mov     DWORD PTR [rcx-12], eax
        dec     r8
```

```
        jne     SHORT $LL3@f
$LN1@f:
        ret     0
f       ENDP
```

Maintenant il y a 3 itérateurs: la variable *cnt* et deux indices, qui sont incrémentés par 12 et 28 à chaque itération. Nous pouvons récrire ce code en C/C++ :

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
        size_t i;
        size_t idx1=0; idx2=0;

        // copier d'un tableau a l'autre selon un schema bizarre
        for (i=0; i<cnt; i++)
        {
                a1[idx1]=a2[idx2];
                idx1+=3;
                idx2+=7;
        };
};
```

Donc, au prix de la mise à jour de 3 itérateurs à chaque itération au lieu d'un, nous pouvons supprimer deux opérations de multiplication.

3.10.2 Deux itérateurs

GCC 4.9 fait encore mieux, en ne laissant que 2 itérateurs:

Listing 3.17 : GCC 4.9 x64 avec optimisation

```
; RDI=a1
; RSI=a2
; RDX=cnt
f:
        test    rdx, rdx  ; cnt==0? sortir si oui
        je      .L1
; calculer l'adresse du dernier élément dans "a2" et la laisser dans RDX
        lea     rax, [0+rdx*4]
; RAX=RDX*4=cnt*4
        sal     rdx, 5
; RDX=RDX<<5=cnt*32
        sub     rdx, rax
; RDX=RDX-RAX=cnt*32-cnt*4=cnt*28
        add     rdx, rsi
; RDX=RDX+RSI=a2+cnt*28
.L3:
        mov     eax, DWORD PTR [rsi]
        add     rsi, 28
        add     rdi, 12
        mov     DWORD PTR [rdi-12], eax
```

```
        cmp     rsi, rdx
        jne     .L3
.L1:
        rep ret
```

Il n'y a plus de variable *counter* : GCC en a conclu qu'elle n'étais pas nécessaire.

Le dernier élément du tableau *a2* est calculé avant le début de la boucle (ce qui est facile: $cnt * 7$) et c'est ainsi que la boucle est arrêtée: itérer jusqu'à ce que le second index atteignent cette valeur pré-calculée.

Vous trouverez plus d'informations sur la multiplication en utilisant des décalages/additions/soustractions ici: 1.24.1 on page 279.

Ce code peut être récrit en C/C++ comme ceci:

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
        size_t idx1=0; idx2=0;
        size_t last_idx2=cnt*7;

        // copier d'un tableau a l'autre selon un schema bizarre
        for (;;)
        {
                a1[idx1]=a2[idx2];
                idx1+=3;
                idx2+=7;
                if (idx2==last_idx2)
                        break;
        };
};
```

GCC (Linaro) 4.9 pour ARM64 fait la même chose, mais il pré-calcule le dernier index de *a1* au lieu de *a2*, ce qui a bien sûr le même effet:

Listing 3.18 : GCC (Linaro) 4.9 ARM64 avec optimisation

```
; X0=a1
; X1=a2
; X2=cnt
f:
        cbz     x2, .L1             ; cnt==0? sortir si oui
; calculer le dernier élément du tableau "a1"
        add     x2, x2, x2, lsl 1
; X2=X2+X2<<1=X2+X2*2=X2*3
        mov     x3, 0
        lsl     x2, x2, 2
; X2=X2<<2=X2*4=X2*3*4=X2*12
.L3:
        ldr     w4, [x1],28         ; charger en X1, ajouter 28 à X1
    (post-incrémentation)
        str     w4, [x0,x3]         ; stocker en X0+X3=a1+X3
        add     x3, x3, 12          ; décaler X3
```

```
        cmp     x3, x2             ; fini?
        bne     .L3
.L1:
        ret
```

GCC 4.4.5 pour MIPS fait la même chose:

Listing 3.19 : GCC 4.4.5 for MIPS avec optimisation (IDA)

```
; $a0=a1
; $a1=a2
; $a2=cnt
f:
; sauter au code de vérification de la boucle:
                beqz    $a2, locret_24
; initialiser le compteur (i) à 0:
                move    $v0, $zero ; slot de délai de branchement, NOP

loc_8:
; charger le mot 32-bit en $a1
                lw      $a3, 0($a1)
; incrémenter le compteur (i):
                addiu   $v0, 1
; vérifier si terminé (comparer "i" dans $v0 et "cnt" dans $a2):
                sltu    $v1, $v0, $a2
; stocker le mot 32-bit en $a0:
                sw      $a3, 0($a0)
; ajouter 0x1C (28) à $a1 à chaque itération:
                addiu   $a1, 0x1C
; sauter au corps de la boulce si i<cnt:
                bnez    $v1, loc_8
; ajouter 0xC (12) à $a0 à chaque itération:
                addiu   $a0, 0xC ; slot de délai de branchement

locret_24:
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

3.10.3 Cas Intel C++ 2011

Les optimisations du compilateur peuvent être bizarre, mais néanmoins toujours correctes. Voici ce qu le compilateur Intel C++ 2011 effectue:

Listing 3.20 : Intel C++ 2011 x64 avec optimisation

```
f       PROC
; parameter 1: rcx = a1
; parameter 2: rdx = a2
; parameter 3: r8 = cnt
.B1.1::
        test      r8, r8
        jbe       exit
```

```
.B1.2::
        cmp      r8, 6
        jbe      just_copy

.B1.3::
        cmp      rcx, rdx
        jbe      .B1.5

.B1.4::
        mov      r10, r8
        mov      r9, rcx
        shl      r10, 5
        lea      rax, QWORD PTR [r8*4]
        sub      r9, rdx
        sub      r10, rax
        cmp      r9, r10
        jge      just_copy2

.B1.5::
        cmp      rdx, rcx
        jbe      just_copy

.B1.6::
        mov      r9, rdx
        lea      rax, QWORD PTR [r8*8]
        sub      r9, rcx
        lea      r10, QWORD PTR [rax+r8*4]
        cmp      r9, r10
        jl       just_copy

just_copy2::
; R8 = cnt
; RDX = a2
; RCX = a1
        xor      r10d, r10d
        xor      r9d, r9d
        xor      eax, eax

.B1.8::
        mov      r11d, DWORD PTR [rax+rdx]
        inc      r10
        mov      DWORD PTR [r9+rcx], r11d
        add      r9, 12
        add      rax, 28
        cmp      r10, r8
        jb       .B1.8
        jmp      exit

just_copy::
; R8 = cnt
; RDX = a2
; RCX = a1
        xor      r10d, r10d
```

```
        xor       r9d, r9d
        xor       eax, eax

.B1.11::
        mov       r11d, DWORD PTR [rax+rdx]
        inc       r10
        mov       DWORD PTR [r9+rcx], r11d
        add       r9, 12
        add       rax, 28
        cmp       r10, r8
        jb        .B1.11

exit::
        ret
```

Tout d'abord, quelques décisions sont prises, puis une des routines est exécutée.

Il semble qu'il teste si les tableaux se recoupent.

C'est une façons très connue d'optimiser les routines de copie de blocs de mémoire. Mais les routines de copie sont les même!

ça doit être une erreur de l'optimiseur Intel C++, qui produit néanmoins un code fonctionnel.

Nous prenons volontairement en compte de tels exemples dans ce livre, afin que lecteur comprenne que le ce que génère un compilateur est parfois bizarre mais toujours correct, car lorsque le compilateur a été testé, il a réussi les tests.

3.11 Duff's device

Le dispositif de Duff[9] est une boucle déroulée avec la possibilité d'y sauter au milieu. La boucle déroulée est implémentée en utilisant une déclaration switch() sans arrêt. Nous allons utiliser ici une version légèrement simplifiée du code original de Tom Duff. Disons que nous voulons écrire une fonction qui efface une zone en mémoire. On pourrait le faire avec une simple boucle, effaçant octet par octet. C'est étonnement lent, puisque tous les ordinateurs modernes ont des bus mémoire bien plus large. Donc, la meilleure façon de faire est d'effacer des zones de mémoire en utilisant des blocs de 4 ou 8 octets. Comme nous allons travailler ici avec un exemple 64-bit, nous allons effacer la mémoire par bloc de 8 octets. Jusqu'ici, tout va bien. Mais qu'en est-il du reste? La routine de mise à zéro de la mémoire peut aussi être appelée pour des zones de taille non multiple de 8. Voici l'algorithme:

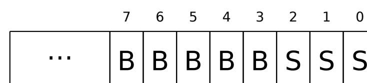
- calculer le nombre de bloc de 8 octets, les effacer en utilisant des accès mémoire 8-octets (64-bit) ;
- calculer la taille du reste, l'effacer en utilisant ces accès mémoire d'un octet.

La seconde étape peut être implémentée en utilisant une simple boucle. Mais implémentons-là avec une boucle déroulée:

[9]Wikipédia

```
#include <stdint.h>
#include <stdio.h>

void bzero(uint8_t* dst, size_t count)
{
        int i;

        if (count&(~7))
                // traiter les blocs de 8 octets
                for (i=0; i<count>>3; i++)
                {
                        *(uint64_t*)dst=0;
                        dst=dst+8;
                };

        // traiter le rset
        switch(count & 7)
        {
        case 7: *dst++ = 0;
        case 6: *dst++ = 0;
        case 5: *dst++ = 0;
        case 4: *dst++ = 0;
        case 3: *dst++ = 0;
        case 2: *dst++ = 0;
        case 1: *dst++ = 0;
        case 0: // ne rien faire
                break;
        }
}
```

Tout d'abord, comprenons comment le calcul est effectué. La taille de la zone mémoire est passée comme une valeur 64-bit. Et cette valeur peut être divisée en deux parties:



(«B » est le nombre de blocs de 8-bit et «S » est la longueur du reste en octets).

Lorsque nous divisons la taille de la zone de mémoire entrée, la valeur est juste décalée de 3 bits vers la droite. Mais pour calculer le reste nous pouvons simplement isoler les 3 bits les plus bas! Donc, le nombre de bloc de 8-octets est calculé comme $count >> 3$ et le reste comme $count\&7$. Nous devons aussi savoir si nous allons exécuter la procédure 8-octets, donc nous devons vérifier si la valeur de $count$ est plus grande que 7. Nous le faisons en mettant à zéro les 3 bits les plus faible et en comparant le résultat avec zéro, car tout ce dont nous avons besoin pour répondre à la question est la partie haute de $count$ non nulle, Bien sûr, ceci fonctionne car 8 est 2^3 et que diviser par des nombres de la forme 2^n est facile. Ce n'est pas possible pour d'autres nombres. Il est difficile de dire si ces astuces valent la peine d'être utilisées, car elles conduisent à du code difficile à lire. Toutefois, ces astuces sont très populaires et un programmeur pratiquant, même s'il/si elle ne va pas les utiliser, doit néanmoins les comprendre. Donc la première partie est simple: obtenir le nombre de blocs de

8-octets et écrire des valeurs 64-bits zéro en mémoire La seconde partie est une boucle déroulée implémentée avec une déclaration switch() sans arrêt.

Premièrement, exprimons en français ce que nous faisons ici.

Nous devons «écrire autant d'octets à zéro en mémoire, que la valeur $count \& 7$ nous l'indique». Si c'est 0, sauter à la fin, et il n'y a rien à faire. Si c'est 1, sauter à l'endroit à l'intérieur de la déclaration switch() où une seule opération de stockage sera exécutée. Si c'est 2, sauter à un autre endroit, où deux opérations de stockage seront exécutées, etc. Une valeur d'entrée de 7 conduit à l'exécution de toutes les 7 opérations. Il n'y a pas de 8, car une zone mémoire de 8 octets serait traitée par la première partie de notre fonction. Donc, nous avons écrit une boucle déroulée. C'était assurément plus rapide sur les anciens ordinateurs que les boucles normales (et au contraire, les CPUs récents travaillent mieux avec des boucles courtes qu'avec des boucles déroulées). Peut-être est-ce encore utile sur les MCU[10]s embarqués moderne à bas coût.

Voyons ce que MSVC 2012 avec optimisation fait:

```
dst$ = 8
count$ = 16
bzero   PROC
        test    rdx, -8
        je      SHORT $LN11@bzero
; traiter les blocs de 8 octets
        xor     r10d, r10d
        mov     r9, rdx
        shr     r9, 3
        mov     r8d, r10d
        test    r9, r9
        je      SHORT $LN11@bzero
        npad    5
$LL19@bzero:
        inc     r8d
        mov     QWORD PTR [rcx], r10
        add     rcx, 8
        movsxd  rax, r8d
        cmp     rax, r9
        jb      SHORT $LL19@bzero
$LN11@bzero:
; traiter le reste
        and     edx, 7
        dec     rdx
        cmp     rdx, 6
        ja      SHORT $LN9@bzero
        lea     r8, OFFSET FLAT:__ImageBase
        mov     eax, DWORD PTR $LN22@bzero[r8+rdx*4]
        add     rax, r8
        jmp     rax
$LN8@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
```

[10]Microcontroller Unit

```
$LN7@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN6@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN5@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN4@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN3@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN2@bzero:
        mov     BYTE PTR [rcx], 0
$LN9@bzero:
        fatret  0
        npad    1
$LN22@bzero:
        DD      $LN2@bzero
        DD      $LN3@bzero
        DD      $LN4@bzero
        DD      $LN5@bzero
        DD      $LN6@bzero
        DD      $LN7@bzero
        DD      $LN8@bzero
bzero   ENDP
```

La premières partie de la fonction est prévisible. La seconde partie est juste une boucle déroulée et un saut y passant le contrôle du flux à la bonne instruction. Il n'y a pas d'autre code entre la paire d'instructions MOV/INC, donc l'exécution va continuer jusqu'à la fin, exécutant autant de paires d'instructions que nécessaire. Á propos, nous pouvons observer que la paire d'instructions MOV/INC utilise un nombre fixe d'octets (3+3). Donc la paire utilise 6 octets. Sachant cela, mous pouvons nous passer de la table des sauts de switch(), nous pouvons simplement multiplier la valeur en entrée par 6 et sauter en $current_RIP + input_value * 6$.

Ceci peut aussi être plus rapide car nous ne devons pas aller chercher une valeur dans la table des sauts.

Il est possible que 6 ne soit pas une très bonne constante pour une multiplication rapide et peut-être que ça n'en vaut pas la peine, mais vous voyez l'idée[11].

C'est ce que les démomakers old-school faisaient dans le passé avec les boucles déroulées.

[11]Comme exercice, vous pouvez essayer de retravailler le code pour se passer de la table des sauts. La paire d'instructions peut être récrite de façon à ce qu'elle utilise 4 octets ou peut-être 8. 1 octet est aussi possible (en utilisant l'instruction STOSB).

3.11.1 Faut-il utiliser des boucles déroulées?

Les boucles déroulées peuvent être bénéfiques si il n'y a pas de cache mémoire rapide entre la RAM et le CPU, et que le CPU, afin d'avoir le code de l'instruction suivante, doit le charger depuis la mémoire à chaque fois. C'est le cas des MCU low-cost moderne et des anciens CPUs.

Les boucles déroulées sont plus lentes que les boucles courtes si il y a un cache rapide entre la RAM et le CPU, et que le corps de la boucle tient dans le cache, et que le CPU va charger le code depuis ce dernier sans toucher à la RAM. Les boucles rapides sont les boucles dont le corps tient dans le cache L1, mais des boucles encore plus rapide sont ces petites qui tiennent dans le cache des micro-opérations.

3.12 Division par la multiplication

Une fonction très simple:

```
int f(int a)
{
        return a/9;
};
```

3.12.1 x86

...est compilée de manière très prédictive:

Listing 3.21 : MSVC

```
_a$ = 8             ; taille = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cdq                ; extension du signe de EAX dans EDX:EAX
    mov    ecx, 9
    idiv   ecx
    pop    ebp
    ret    0
_f   ENDP
```

IDIV divise le nombre 64-bit stocké dans la paire de registres EDX:EAX par la valeur dans ECX. Comme résultat, EAX contiendra le quotient, et EDX— le reste. Le résultat de la fonction f() est renvoyé dans le registre EAX, donc la valeur n'est pas déplacée après la division, elle est déjà à la bonne place.

Puisque IDIV utilise la valeur dans la paire de registres EDX:EAX, l'instruction CDQ (avant IDIV) étend la valeur dans EAX en une valeur 64-bit, en tenant compte du signe, tout comme MOVSX le fait.

Si nous mettons l'optimisation (/Ox), nous obtenons:

Listing 3.22 : MSVC avec optimisation

```
_a$ = 8                      ; size = 4
_f    PROC

    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, 954437177   ; 38e38e39H
    imul   ecx
    sar    edx, 1
    mov    eax, edx
    shr    eax, 31          ; 0000001fH
    add    eax, edx
    ret    0
_f    ENDP
```

Ceci est la division par la multiplication. L'opération de multiplication est bien plus rapide. Et il possible d'utiliser cette astuce [12] pour produire du code effectivement équivalent et plus rapide.

Ceci est aussi appelé «strength reduction» dans les optimisations du compilateur.

GCC 4.4.1 génère presque le même code, même sans flag d'optimisation, tout comme MSVC avec l'optimisation:

Listing 3.23 : GCC 4.4.1 sans optimisation

```
       public f
f      proc near

arg_0  = dword ptr  8

       push    ebp
       mov     ebp, esp
       mov     ecx, [ebp+arg_0]
       mov     edx, 954437177 ; 38E38E39h
       mov     eax, ecx
       imul    edx
       sar     edx, 1
       mov     eax, ecx
       sar     eax, 1Fh
       mov     ecx, edx
       sub     ecx, eax
       mov     eax, ecx
       pop     ebp
       retn
f      endp
```

3.12.2 Comment ça marche

Des mathématiques du niveau de l'école, nous pouvons nous souvenir que la division par 9 peut être remplacée par la multiplication par $\frac{1}{9}$. En fait, parfois les compilateurs font cela pour l'arithmétique en virgule flottante, par exemple, l'instruction FDIV en

[12] En savoir plus sur la division par la multiplication dans [Henry S. Warren, *Hacker's Delight*, (2002)10-3]

code x86 peut être remplacée par FMUL. Au moins MSVC 6.0 va remplacer la division par 9 par un multiplication par $0.111111...$ et parfois il est difficile d'être sûr de quelle opération il s'agissait dans le code source.

Mais lorsque nous opérons avec des valeurs entières et des registres CPU entier, nous ne pouvons pas utiliser de fractions. Toutefois, nous pouvons retravailler la fraction comme ceci:

$$result = \frac{x}{9} = x \cdot \frac{1}{9} = x \cdot \frac{1 \cdot MagicNumber}{9 \cdot MagicNumber}$$

Avec le fait que la division par 2^n est très rapide (en utilisant des décalages), nous devons maintenant trouver quels $MagicNumber$, pour lesquels l'équation suivante sera vraie: $2^n = 9 \cdot MagicNumber$.

La division par 2^{32} est quelque peu cachée: la partie basse 32-bit du produit dans EAX n'est pas utilisée (ignorée), seule la partie haute 32-bit du produit (dans EDX) est utilisée et ensuite décalée de 1 bit additionnel.

Autrement dit, le code assembleur que nous venons de voir multiplie par $\frac{954437177}{2^{32+1}}$, ou divise par $\frac{2^{32+1}}{954437177}$. Pour trouver le diviseur, nous avons juste à diviser le numérateur par le dénominateur. En utilisant Wolfram Alpha, nous obtenons 8.99999999.... comme résultat (qui est proche de 9).

En lire plus à ce sujet dans [Henry S. Warren, *Hacker's Delight*, (2002)10-3].

Beaucoup de gens manquent la division “cachée” par 2^{32} or 2^{64}, lorsque la partie basse 32-bit (ou la partie 64-bit) du produit n'est pas utilisée. C'est pourquoi la division par la multiplication est difficile à comprendre au début.

Mathematical Recipes[13]a une autre explication.

3.12.3 ARM

Le processeur ARM, tout comme un autre processeur «pur» RISC n'a pas d'instruction pour la division. Il manque aussi une simple instruction pour la multiplication avec une constante 32-bit (rappelez-vous qu'une constante 32-bit ne tient pas dans un opcode 32-bit).

En utilisant de cette astuce intelligente (ou *hack*), il est possible d'effectuer la division en utilisant seulement trois instructions: addition, soustraction et décalages de bit (1.28 on page 392).

Voici un exemple qui divise un nombre 32-bit par 10, tiré de [Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)3.3 Division by a Constant]. La sortie est constituée du quotient et du reste.

```
; prend l'argument dans a1
; renvoie le quotient dans a1, le reste dans a2
; on peut utiliser moins de cycles si seul le quotient ou le reste est requis
    SUB    a2, a1, #10            ; garde (x-10) pour plus tard
    SUB    a1, a1, a1, lsr #2
    ADD    a1, a1, a1, lsr #4
```

[13] https://math.recipes

```
    ADD    a1, a1, a1, lsr #8
    ADD    a1, a1, a1, lsr #16
    MOV    a1, a1, lsr #3
    ADD    a3, a1, a1, asl #2
    SUBS   a2, a2, a3, asl #1      ; calcule (x-10) - (x/10)*10
    ADDPL  a1, a1, #1              ; fix-up quotient
    ADDMI  a2, a2, #10             ; fix-up reste
    MOV    pc, lr
```

avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
__text:00002C58 39 1E 08 E3 E3 18 43 E3  MOV    R1, 0x38E38E39
__text:00002C60 10 F1 50 E7              SMMUL  R0, R0, R1
__text:00002C64 C0 10 A0 E1              MOV    R1, R0,ASR#1
__text:00002C68 A0 0F 81 E0              ADD    R0, R1, R0,LSR#31
__text:00002C6C 1E FF 2F E1              BX     LR
```

Ce code est presque le même que celui généré par MSVC avec optimisation et GCC.

Il semble que LLVM utilise le même algorithme pour générer des constantes.

Le lecteur attentif pourrait se demander comment MOV écrit une valeur 32-bit dans un registre, alors que ceci n'est pas possible en mode ARM.

C'est impossible, en effet, mais on voit qu'il y a 8 octets par instruction, au lieu des 4 standards, en fait, ce sont deux instructions.

La première instruction charge 0x8E39 dans les 16 bits bas du registre et la seconde instruction est MOVT, qui charge 0x383E dans les 16 bits hauts du registre. IDA reconnaît de telles séquences, et par concision, il les réduit a une seule «pseudo-instruction».

L'instruction SMMUL (*Signed Most Significant Word Multiply* mot le plus significatif d'une multiplication signée), multiplie deux nombres, les traitant comme des nombres signés et laisse la partie 32-bit haute dans le registre R0, en ignorant la partie 32-bit basse du résultat.

L'instruction «MOV R1, R0,ASR#1» est le décalage arithmétique à droite d'un bit.

«ADD R0, R1, R0,LSR#31» est $R0 = R1 + R0 >> 31$

Il n'y a pas d'instruction de décalage séparée en mode ARM. A la place, des instructions comme (MOV, ADD, SUB, RSB)[14] peuvent avoir un suffixe, indiquant si le second argument doit être décalé, et si oui, de quelle valeur et comment. ASR signifie *Arithmetic Shift Right*, LSR—*Logical Shift Right*.

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
MOV             R1, 0x38E38E39
SMMUL.W         R0, R0, R1
ASRS            R1, R0, #1
```

[14]Ces instructions sont également appelées «instructions de traitement de données»

```
ADD.W           R0, R1, R0,LSR#31
BX              LR
```

Il y a dex instructions de décalage séparées en mode Thumb, et l'une d'elle est utilisée ici—ASRS (arithmetic shift right).

sans optimisation Xcode 4.6.3 (LLVM) and Keil 6/2013

LLVM sans optimisation ne génère pas le code que nous avons vu avant dans cette section, mais insère à la place un appel à la fonction de bibliothèque *___divsi3*.

À propos de Keil: il insère un appel à la fonction de bibliothèque *__aeabi_idivmod* dans tous les cas.

3.12.4 MIPS

Pour une raison quelconque, GCC 4.4.5 avec optimisation génère seulement une instruction de division:

Listing 3.24 : avec optimisation GCC 4.4.5 (IDA)

```
f:
                li      $v0, 9
                bnez    $v0, loc_10
                div     $a0, $v0 ; slot de délai de branchement
                break   0x1C00   ; "break 7" en assembleur sortir et objdump

loc_10:
                mflo    $v0
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

Ici, nous voyons une nouvelle instruction: BREAK. Elle lève simplement une exception.

Dans ce cas, une exception est levée si le diviseur est zéro (il n'est pas possible de diviser par zéro dans les mathématiques conventionnelles).

Mais GCC n'a probablement pas fait correctement le travail d'optimisation et n'a pas vu que $V0 ne vaut jamais zéro.

Donc le test est laissé ici. Donc, si $V0 est zéro, BREAK est exécuté, signalant l'exception à l'OS.

Autrement, MFLO s'exécute, qui prend le résultat de la division depuis le registre LO et le copie dans $V0.

À propos, comme on devrait le savoir, l'instruction MUL laisse les 32bits hauts du résultat dans le registre HI et les 32 bits bas dans le registre LO.

DIV laisse le résultat dans le registre LO, et le reste dans le registre HI.

Si nous modifions la déclaration en «a % 9», l'instruction MFHI est utilisée au lieu de MFLO.

3.12.5 Exercice

- http://challenges.re/27

3.13 Conversion de chaîne en nombre (atoi())

Essayons de ré-implémenter la fonction C standard atoi().

3.13.1 Exemple simple

Voici la manière la plus simple possible de lire un nombre encodé en ASCII.

C'est sujet aux erreurs: un caractère autre qu'un nombre conduit à un résultat incorrect.

```
#include <stdio.h>

int my_atoi (char *s)
{
        int rt=0;

        while (*s)
        {
                rt=rt*10 + (*s-'0');
                s++;
        };

        return rt;
};

int main()
{
        printf ("%d\n", my_atoi ("1234"));
        printf ("%d\n", my_atoi ("1234567890"));
};
```

Donc, tout ce que fait l'algorithme, c'est de lire les chiffres de gauche à droite.

Le caractère ASCII zéro est soustrait de chaque chiffre.

Les chiffres de «0» à «9» sont consécutifs dans la table ASCII, donc nous n'avons même pas besoin de connaître la valeur exacte du caractère «0».

Tout ce que nous avons besoin de savoir, c'est que «0» moins «0» vaut 0, «9» moins «0» vaut 9, et ainsi de suite.

Soustraire «0» de chaque caractère résulte en un nombre de 0 à 9 inclus.

Tout autre caractère conduit à un résultat incorrect, bien sûr!

Chaque chiffre doit être ajouté au résultat final (dans la variable «rt»), mais le résultat final est aussi multiplié par 10 à chaque chiffre.

Autrement dit, le résultat est décalé à gauche d'une position au format décimal à chaque itération.

Le dernier chiffre est ajouté, mais il n'y a pas de décalage.

MSVC 2013 x64 avec optimisation

Listing 3.25 : MSVC 2013 x64 avec optimisation

```
s$ = 8
my_atoi PROC
; charger le premier caractère
        movzx   r8d, BYTE PTR [rcx]
; EAX est alloué pour la variable "rt"
; qui vaut 0 au début
        xor     eax, eax
; est-ce que le premier caractère est un octet à zéro, i.e., fin de chaine?
; si oui, sortir
        test    r8b, r8b
        je      SHORT $LN9@my_atoi
$LL2@my_atoi:
        lea     edx, DWORD PTR [rax+rax*4]
; EDX=RAX+RAX*4=rt+rt*4=rt*5
        movsx   eax, r8b
; EAX=caractère en entrée
; charger le caractère suivant dans R8D
        movzx   r8d, BYTE PTR [rcx+1]
; décaler le pointeur dans RCX sur le caractère suivant:
        lea     rcx, QWORD PTR [rcx+1]
        lea     eax, DWORD PTR [rax+rdx*2]
; EAX=RAX+RDX*2=caractère en entrée + rt*5*2=caractère en entrée + rt*10
: chiffre correct en soustrayant 48 (0x30 ou '0')
        add     eax, -48                                ; ffffffffffffffd0H
; est-ce que le dernier caractère était zéro?
        test    r8b, r8b
; sauter au début de la boucle si non
        jne     SHORT $LL2@my_atoi
$LN9@my_atoi:
        ret     0
my_atoi ENDP
```

Un caractère peut-être chargé à deux endroits: le premier caractère et tous les caractères subséquents. Ceci est arrangé de cette manière afin de regrouper les boucles.

Il n'y a pas d'instructions pour multiplier par 10, à la place, deux instructions LEA le font.

Parfois, MSVC utilise l'instruction ADD avec une constante négative à la place d'un SUB. C'est le cas.

C'est très difficile de dire pourquoi c'est meilleur que SUB. Mais MSVC fait souvent ceci.

GCC 4.9.1 x64 avec optimisation

GCC 4.9.1 avec optimisation est plus concis, mais il y a une instruction RET redondante à la fin. Une suffit.

Listing 3.26 : GCC 4.9.1 x64 avec optimisation

```
my_atoi:
; charger le caractère en entrée dans EDX
        movsx   edx, BYTE PTR [rdi]
; EAX est alloué pour la variable "rt"
        xor     eax, eax
; sortir, si le caractère chargé est l'octet nul
        test    dl, dl
        je      .L4
.L3:
        lea     eax, [rax+rax*4]
; EAX=RAX*5=rt*5
; décaler le pointeur sur le caractère suivant:
        add     rdi, 1
        lea     eax, [rdx-48+rax*2]
; EAX=caractère en entrée - 48 + RAX*2 = caractère en entrée - '0' + rt*10
; charger le caractère suivant:
        movsx   edx, BYTE PTR [rdi]
; sauter au début de la boucle, si le caractère chargé n'est pas l'octet nul
        test    dl, dl
        jne     .L3
        rep ret
.L4:
        rep ret
```

avec optimisation Keil 6/2013 (Mode ARM)

Listing 3.27 : avec optimisation Keil 6/2013 (Mode ARM)

```
my_atoi PROC
; R1 contiendra le pointeur sur le caractère
        MOV      r1,r0
; R0 contiendra la variable "rt"
        MOV      r0,#0
        B        |L0.28|
|L0.12|
        ADD      r0,r0,r0,LSL #2
; R0=R0+R0<<2=rt*5
        ADD      r0,r2,r0,LSL #1
; R0=caractère entré +rt*5<<1 = caractère entré + rt*10
; corriger le tout en soustrayant '0' de rt:
        SUB      r0,r0,#0x30
; décaler le pointeur sur le caractère suivant:
        ADD      r1,r1,#1
|L0.28|
; charger le caractère entré dans R2
        LDRB     r2,[r1,#0]
; est-ce que c'est l'octet nul? si non, sauter au corps de la boucle.
```

```
        CMP      r2,#0
        BNE      |L0.12|
; sortir si octet nul.
; la variable "rt" est encore dans le registre R0, prête à être utilisée dans
; la fonction appelante
        BX       lr
        ENDP
```

avec optimisation Keil 6/2013 (Mode Thumb)

Listing 3.28 : avec optimisation Keil 6/2013 (Mode Thumb)

```
my_atoi PROC
; R1 est le pointeur sur le caractère en entrée
        MOVS     r1,r0
; R0 est alloué pour la variable "rt"
        MOVS     r0,#0
        B        |L0.16|
|L0.6|
        MOVS     r3,#0xa
; R3=10
        MULS     r0,r3,r0
; R0=R3*R0=rt*10
; décaler le pointeur sur le caractère suivant:
        ADDS     r1,r1,#1
: corriger le tout en lui soustrayant le caractère '0':
        SUBS     r0,r0,#0x30
        ADDS     r0,r2,r0
; rt=R2+R0=caractère entré + (rt*10 - '0')
|L0.16|
; charger le caractère entré dans R2
        LDRB     r2,[r1,#0]
; est-ce zéro?
        CMP      r2,#0
; sauter au corps de la boucle si non
        BNE      |L0.6|
; la variable rt est maintenant dans R0, prête a être utilisé dans la
    fonction appelante
        BX       lr
        ENDP
```

De façon intéressante, nous pouvons nous rappeler des cours de mathématiques que l'ordre des opérations d'addition et de soustraction n'a pas d'importance.

C'est notre cas: d'abord, l'expression $rt * 10 -' 0'$ est calculée, puis la valeur du caractère en entrée lui est ajoutée.

En effet, le résultat est le même, mais le compilateur a fait quelques regroupements.

GCC 4.9.1 ARM64 avec optimisation

Le compilateur ARM64 peut utiliser le suffixe de pré-incrémentation pour les instructions:

Listing 3.29 : GCC 4.9.1 ARM64 avec optimisation

```
my_atoi:
; charger le caractère en entrée dans W1
        ldrb    w1, [x0]
        mov     x2, x0
; X2=adresse de la chaîne en entrée
; est-ce que le caractère chargé est zéro?
: sauter à la sortie si oui
; W1 contiendra 0 dans ce cas
; il sera rechargé dans W0 en L4.
        cbz     w1, .L4
; W0 contiendra la variable "rt"
; initialisons-la à zéro
        mov     w0, 0
.L3:
; soustraire 48 ou '0' de la variable en entrée et mettre le résultat dans
    W3:
        sub     w3, w1, #48
; charger le caractère suivant à l'adresse X2+1 dans W1 avec
    pré-incrémentation:
        ldrb    w1, [x2,1]!
        add     w0, w0, w0, lsl 2
; W0=W0+W0<<2=W0+W0*4=rt*5
        add     w0, w3, w0, lsl 1
; W0=chiffre entrée + W0<<1 = chiffre entrée + rt*5*2 = chiffre entrée +
    rt*10
; si le caractère que nous venons de charger n'est pas l'octet nul,
; sauter au début de la boucle
        cbnz    w1, .L3
; la variable qui doit être retournée (rt) est dans W0, prête à être utilisée
; dans la fonction appelante
        ret
.L4:
        mov     w0, w1
        ret
```

3.13.2 Un exemple légèrement avancé

Mon nouvel extrait de code est plus avancé, maintenant il teste le signe «moins» au premier caractère et renvoie une erreur si un caractère autre qu'un chiffre est trouvé dans la chaîne en entrée:

```
#include <stdio.h>

int my_atoi (char *s)
{
        int negative=0;
        int rt=0;

        if (*s=='-')
        {
                negative=1;
                s++;
```

```
        };

        while (*s)
        {
                if (*s<'0' || *s>'9')
                {
                        printf ("Error! Unexpected char: '%c'\n", *s);
                        exit(0);
                };
                rt=rt*10 + (*s-'0');
                s++;
        };

        if (negative)
                return -rt;
        return rt;
};

int main()
{
        printf ("%d\n", my_atoi ("1234"));
        printf ("%d\n", my_atoi ("1234567890"));
        printf ("%d\n", my_atoi ("-1234"));
        printf ("%d\n", my_atoi ("-1234567890"));
        printf ("%d\n", my_atoi ("-a1234567890")); // error
};
```

GCC 4.9.1 x64 avec optimisation

Listing 3.30 : GCC 4.9.1 x64 avec optimisation

```
.LC0:
        .string "Error! Unexpected char: '%c'\n"

my_atoi:
        sub     rsp, 8
        movsx   edx, BYTE PTR [rdi]
; tester si c'est le signe moins
        cmp     dl, 45 ; '-'
        je      .L22
        xor     esi, esi
        test    dl, dl
        je      .L20
.L10:
; ESI=0 ici si il n'y avait pas de signe moins et 1 si il en avait un
        lea     eax, [rdx-48]
; tout caractère autre qu'un chiffre résultera en
; un nombre non signé plus grand que 9 après
; soustraction donc s'il ne s'agit pas d'un chiffre,
; sauter en L4, où l'erreur doit être rapportée
        cmp     al, 9
        ja      .L4
        xor     eax, eax
```

```
        jmp     .L6
.L7:
        lea     ecx, [rdx-48]
        cmp     cl, 9
        ja      .L4
.L6:
        lea     eax, [rax+rax*4]
        add     rdi, 1
        lea     eax, [rdx-48+rax*2]
        movsx   edx, BYTE PTR [rdi]
        test    dl, dl
        jne     .L7
; s'il n'y avait pas de signe moins, sauter l'instruction NEG
; s'il y en avait un, l'exécuter.
        test    esi, esi
        je      .L18
        neg     eax
.L18:
        add     rsp, 8
        ret
.L22:
        movsx   edx, BYTE PTR [rdi+1]
        lea     rax, [rdi+1]
        test    dl, dl
        je      .L20
        mov     rdi, rax
        mov     esi, 1
        jmp     .L10
.L20:
        xor     eax, eax
        jmp     .L18
.L4:
; signale une erreur. le caractère est dans EDX
        mov     edi, 1
        mov     esi, OFFSET FLAT:.LC0 ; "Error! Unexpected char: '%c'\n"
        xor     eax, eax
        call    __printf_chk
        xor     edi, edi
        call    exit
```

Si le signe «moins» a été rencontré au début de la chaîne, l'instruction NEG est exécutée à la fin. Elle rend le nombre négatif.

Il y a encore une chose à mentionner.

Comment ferait un programmeur moyen pour tester si le caractère n'est pas un chiffre? Tout comme nous l'avons dans le code source:

```
if (*s<'0' || *s>'9')
    ...
```

Il y a deux opérations de comparaison.

Ce qui est intéressant, c'est que l'on peut remplacer les deux opérations par une seule: simplement soustraire «0» de la valeur du caractère,

traiter le résultat comme une valeur non-signée (ceci est important) et tester s'il est plus grand que 9.

Par exemple, disons que l'entrée utilisateur contient le caractère point («.») qui a pour code ASCII 46. $46 - 48 = -2$ si nous traitons le résultat comme un nombre signé.

En effet, le caractère point est situé deux places avant le caractère «0» dans la table ASCII. Mais il correspond à 0xFFFFFFFE (4294967294) si nous traitons le résultat comme une valeur non signée, c'est définitivement plus grand que 9!

Le compilateur fait cela souvent, donc il est important de connaître ces astuces.

Un autre exemple dans ce livre: 3.19.1 on page 688.

MSVC 2013 x64 avec optimisation utilise les même astuces.

avec optimisation Keil 6/2013 (Mode ARM)

Listing 3.31 : avec optimisation Keil 6/2013 (Mode ARM)

```
my_atoi PROC
        PUSH     {r4-r6,lr}
        MOV      r4,r0
        LDRB     r0,[r0,#0]
        MOV      r6,#0
        MOV      r5,r6
        CMP      r0,#0x2d '-'
; R6 contiendra 1 si le signe moins a été rencontré, 0 sinon
        MOVEQ    r6,#1
        ADDEQ    r4,r4,#1
        B        |L0.80|
|L0.36|
        SUB      r0,r1,#0x30
        CMP      r0,#0xa
        BCC      |L0.64|
        ADR      r0,|L0.220|
        BL       __2printf
        MOV      r0,#0
        BL       exit
|L0.64|
        LDRB     r0,[r4],#1
        ADD      r1,r5,r5,LSL #2
        ADD      r0,r0,r1,LSL #1
        SUB      r5,r0,#0x30
|L0.80|
        LDRB     r1,[r4,#0]
        CMP      r1,#0
        BNE      |L0.36|
        CMP      r6,#0
; negate result
        RSBNE    r0,r5,#0
        MOVEQ    r0,r5
```

```
33          POP      {r4-r6,pc}
34          ENDP
35
36  |L0.220|
37          DCB      "Error! Unexpected char: '%c'\n",0
```

Il n'y a pas d'instruction NEG en ARM 32-bit, donc l'opération «Reverse Subtraction» (ligne 31) est utilisée ici.

Elle est déclenchée si le résultat de l'instruction CMP (à la ligne 29) était «Not Equal» (non égal) (d'où le suffixe -NE).

Donc ce que fait RSBNE, c'est soustraire la valeur résultante de 0.

Cela fonctionne comme l'opération de soustraction normale, mais échange les opérandes

Soustraire n'importe quel nombre de 0 donne sa négation: $0 - x = -x$.

Le code en mode Thumb est en gros le même.

GCC 4.9 pour ARM64 peut utiliser l'instruction NEG, qui est disponible en ARM64.

3.13.3 Exercice

Oh, à propos, les chercheurs en sécurité sont souvent confrontés à un comportement imprévisible de programme lorsqu'il traite des données incorrectes.

Par exemple, lors du fuzzing. À titre d'exercice, vous pouvez essayer d'entrer des caractères qui ne soient pas des chiffres et de voir ce qui se passe.

Essayez d'expliquer ce qui s'est passé, et pourquoi.

3.14 Fonctions inline

Le code inline, c'est lorsque le compilateur, au lieu de mettre une instruction d'appel à une petite ou à une minuscule fonction, copie son corps à la place.

Listing 3.32 : Un exemple simple

```
#include <stdio.h>

int celsius_to_fahrenheit (int celsius)
{
        return celsius * 9 / 5 + 32;
};

int main(int argc, char *argv[])
{
        int celsius=atol(argv[1]);
        printf ("%d\n", celsius_to_fahrenheit (celsius));
};
```

...est compilée de façon très prédictive, toutefois, si nous utilisons l'option d'optimisation de GCC (-O3), nous voyons:

Listing 3.33 : GCC 4.8.1 avec optimisation

```
_main:
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        call    ___main
        mov     eax, DWORD PTR [ebp+12]
        mov     eax, DWORD PTR [eax+4]
        mov     DWORD PTR [esp], eax
        call    _atol
        mov     edx, 1717986919
        mov     DWORD PTR [esp], OFFSET FLAT:LC2 ; "%d\n"
        lea     ecx, [eax+eax*8]
        mov     eax, ecx
        imul    edx
        sar     ecx, 31
        sar     edx
        sub     edx, ecx
        add     edx, 32
        mov     DWORD PTR [esp+4], edx
        call    _printf
        leave
        ret
```

(Ici la division est effectuée avec une multiplication(3.12 on page 636).)

Oui, notre petite fonction `celsius_to_fahrenheit()` a été placée juste avant l'appel à `printf()`.

Pourquoi? C'est plus rapide que d'exécuter la code de cette fonction plus le surcoût de l'appel/retour.

Les optimiseurs des compilateurs modernes choisissent de mettre en ligne les petites fonctions automatiquement. Mais il est possible de forcer le compilateur à mettre en ligne automatiquement certaines fonctions, en les marquants avec le mot clef «inline» dans sa déclaration.

3.14.1 Fonctions de chaînes et de mémoire

Une autre tactique courante d'optimisation automatique est la mise en ligne des fonctions de chaînes comme *strcpy()*, *strcmp()*, *strlen()*, *memset()*, *memcmp()*, *memcpy()*, etc..

Parfois, c'est plus rapide que d'appeler une fonction séparée.

Ce sont des patterns très fréquents et il est hautement recommandé aux rétro-ingénieurs d'apprendre à les détecter automatiquement.

strcmp()

Listing 3.34 : exemple strcmp()

```
bool is_bool (char *s)
{
        if (strcmp (s, "true")==0)
                return true;
        if (strcmp (s, "false")==0)
                return false;

        assert(0);
};
```

Listing 3.35 : avec optimisation GCC 4.8.1

```
.LC0:
        .string "true"
.LC1:
        .string "false"
is_bool:
.LFB0:
        push    edi
        mov     ecx, 5
        push    esi
        mov     edi, OFFSET FLAT:.LC0
        sub     esp, 20
        mov     esi, DWORD PTR [esp+32]
        repz cmpsb
        je      .L3
        mov     esi, DWORD PTR [esp+32]
        mov     ecx, 6
        mov     edi, OFFSET FLAT:.LC1
        repz cmpsb
        seta    cl
        setb    dl
        xor     eax, eax
        cmp     cl, dl
        jne     .L8
        add     esp, 20
        pop     esi
        pop     edi
        ret
.L8:
        mov     DWORD PTR [esp], 0
        call    assert
        add     esp, 20
        pop     esi
        pop     edi
        ret
.L3:
        add     esp, 20
        mov     eax, 1
        pop     esi
```

```
        pop     edi
        ret
```

Listing 3.36 : MSVC 2010 avec optimisation

```
$SG3454 DB      'true', 00H
$SG3456 DB      'false', 00H

_s$ = 8         ; taille = 4
?is_bool@@YA_NPAD@Z PROC ; is_bool
        push    esi
        mov     esi, DWORD PTR _s$[esp]
        mov     ecx, OFFSET $SG3454 ; 'true'
        mov     eax, esi
        npad    4 ; aligner le label suivant
$LL6@is_bool:
        mov     dl, BYTE PTR [eax]
        cmp     dl, BYTE PTR [ecx]
        jne     SHORT $LN7@is_bool
        test    dl, dl
        je      SHORT $LN8@is_bool
        mov     dl, BYTE PTR [eax+1]
        cmp     dl, BYTE PTR [ecx+1]
        jne     SHORT $LN7@is_bool
        add     eax, 2
        add     ecx, 2
        test    dl, dl
        jne     SHORT $LL6@is_bool
$LN8@is_bool:
        xor     eax, eax
        jmp     SHORT $LN9@is_bool
$LN7@is_bool:
        sbb     eax, eax
        sbb     eax, -1
$LN9@is_bool:
        test    eax, eax
        jne     SHORT $LN2@is_bool

        mov     al, 1
        pop     esi

        ret     0
$LN2@is_bool:

        mov     ecx, OFFSET $SG3456 ; 'false'
        mov     eax, esi
$LL10@is_bool:
        mov     dl, BYTE PTR [eax]
        cmp     dl, BYTE PTR [ecx]
        jne     SHORT $LN11@is_bool
        test    dl, dl
        je      SHORT $LN12@is_bool
        mov     dl, BYTE PTR [eax+1]
        cmp     dl, BYTE PTR [ecx+1]
```

```
        jne     SHORT $LN11@is_bool
        add     eax, 2
        add     ecx, 2
        test    dl, dl
        jne     SHORT $LL10@is_bool
$LN12@is_bool:
        xor     eax, eax
        jmp     SHORT $LN13@is_bool
$LN11@is_bool:
        sbb     eax, eax
        sbb     eax, -1
$LN13@is_bool:
        test    eax, eax
        jne     SHORT $LN1@is_bool

        xor     al, al
        pop     esi

        ret     0
$LN1@is_bool:

        push    11
        push    OFFSET $SG3458
        push    OFFSET $SG3459
        call    DWORD PTR __imp___wassert
        add     esp, 12
        pop     esi

        ret     0
?is_bool@@YA_NPAD@Z ENDP ; is_bool
```

strlen()

Listing 3.37 : exemple strlen()

```
int strlen_test(char *s1)
{
        return strlen(s1);
};
```

Listing 3.38 : avec optimisation MSVC 2010

```
_s1$ = 8 ; size = 4
_strlen_test PROC
        mov     eax, DWORD PTR _s1$[esp-4]
        lea     edx, DWORD PTR [eax+1]
$LL3@strlen_tes:
        mov     cl, BYTE PTR [eax]
        inc     eax
        test    cl, cl
        jne     SHORT $LL3@strlen_tes
        sub     eax, edx
        ret     0
```

```
_strlen_test ENDP
```

strcpy()

Listing 3.39 : exemple strcpy()

```
void strcpy_test(char *s1, char *outbuf)
{
        strcpy(outbuf, s1);
};
```

Listing 3.40 : avec optimisation MSVC 2010

```
_s1$ = 8        ; taille = 4
_outbuf$ = 12   ; taille = 4
_strcpy_test PROC
        mov     eax, DWORD PTR _s1$[esp-4]
        mov     edx, DWORD PTR _outbuf$[esp-4]
        sub     edx, eax
        npad    6 ; aligner le label suivant
$LL3@strcpy_tes:
        mov     cl, BYTE PTR [eax]
        mov     BYTE PTR [edx+eax], cl
        inc     eax
        test    cl, cl
        jne     SHORT $LL3@strcpy_tes
        ret     0
_strcpy_test ENDP
```

memset()

Exemple#1

Listing 3.41 : 32 bytes

```
#include <stdio.h>

void f(char *out)
{
        memset(out, 0, 32);
};
```

De nombreux compilateurs ne génèrent pas un appel à memset() pour de petits blocs, mais insèrent plutôt un paquet de MOVs:

Listing 3.42 : GCC 4.9.1 x64 avec optimisation

```
f:
        mov     QWORD PTR [rdi], 0
        mov     QWORD PTR [rdi+8], 0
        mov     QWORD PTR [rdi+16], 0
```

```
        mov     QWORD PTR [rdi+24], 0
        ret
```

À propos, ça nous rappelle le déroulement de boucles: 1.22.1 on page 251.

Exemple#2

Listing 3.43 : 67 bytes

```
#include <stdio.h>

void f(char *out)
{
        memset(out, 0, 67);
};
```

Lorsque la taille du bloc n'est pas un multiple de 4 ou 8, les compilateurs se comportent différemment.

Par exemple, MSVC 2012 continue à insérer des MOVs:

Listing 3.44 : MSVC 2012 x64 avec optimisation

```
out$ = 8
f       PROC
        xor     eax, eax
        mov     QWORD PTR [rcx], rax
        mov     QWORD PTR [rcx+8], rax
        mov     QWORD PTR [rcx+16], rax
        mov     QWORD PTR [rcx+24], rax
        mov     QWORD PTR [rcx+32], rax
        mov     QWORD PTR [rcx+40], rax
        mov     QWORD PTR [rcx+48], rax
        mov     QWORD PTR [rcx+56], rax
        mov     WORD PTR [rcx+64], ax
        mov     BYTE PTR [rcx+66], al
        ret     0
f       ENDP
```

…tandis que GCC utilise REP STOSQ, en concluant que cela sera plus petit qu'un paquet de MOVs:

Listing 3.45 : GCC 4.9.1 x64 avec optimisation

```
f:
        mov     QWORD PTR [rdi], 0
        mov     QWORD PTR [rdi+59], 0
        mov     rcx, rdi
        lea     rdi, [rdi+8]
        xor     eax, eax
        and     rdi, -8
        sub     rcx, rdi
        add     ecx, 67
```

```
        shr     ecx, 3
        rep stosq
        ret
```

memcpy()

Petits blocs

La routine pour copier des blocs courts est souvent implémentée comme une séquence d'instructions MOV.

Listing 3.46 : exemple memcpy()

```
void memcpy_7(char *inbuf, char *outbuf)
{
        memcpy(outbuf+10, inbuf, 7);
};
```

Listing 3.47 : MSVC 2010 avec optimisation

```
_inbuf$ = 8     ; size = 4
_outbuf$ = 12   ; size = 4
_memcpy_7 PROC
        mov     ecx, DWORD PTR _inbuf$[esp-4]
        mov     edx, DWORD PTR [ecx]
        mov     eax, DWORD PTR _outbuf$[esp-4]
        mov     DWORD PTR [eax+10], edx
        mov     dx, WORD PTR [ecx+4]
        mov     WORD PTR [eax+14], dx
        mov     cl, BYTE PTR [ecx+6]
        mov     BYTE PTR [eax+16], cl
        ret     0
_memcpy_7 ENDP
```

Listing 3.48 : GCC 4.8.1 avec optimisation

```
memcpy_7:
        push    ebx
        mov     eax, DWORD PTR [esp+8]
        mov     ecx, DWORD PTR [esp+12]
        mov     ebx, DWORD PTR [eax]
        lea     edx, [ecx+10]
        mov     DWORD PTR [ecx+10], ebx
        movzx   ecx, WORD PTR [eax+4]
        mov     WORD PTR [edx+4], cx
        movzx   eax, BYTE PTR [eax+6]
        mov     BYTE PTR [edx+6], al
        pop     ebx
        ret
```

C'est effectué en général ainsi: des blocs de 4-octets sont d'abord copiés, puis un mot de 16-bit (si nécessaire) et enfin un dernier octet (si nécessaire).

Les structures sont aussi copiées en utilisant MOV : 1.30.4 on page 466.

Longs blocs

Les compilateurs se comportent différemment dans ce cas.

Listing 3.49 : memcpy() exemple

```
void memcpy_128(char *inbuf, char *outbuf)
{
        memcpy(outbuf+10, inbuf, 128);
};

void memcpy_123(char *inbuf, char *outbuf)
{
        memcpy(outbuf+10, inbuf, 123);
};
```

Pour copier 128 octets, MSVC utilise une seule instruction MOVSD (car 128 est divisible par 4) :

Listing 3.50 : MSVC 2010 avec optimisation

```
_inbuf$ = 8             ; size = 4
_outbuf$ = 12           ; size = 4
_memcpy_128 PROC
        push    esi
        mov     esi, DWORD PTR _inbuf$[esp]
        push    edi
        mov     edi, DWORD PTR _outbuf$[esp+4]
        add     edi, 10
        mov     ecx, 32
        rep movsd
        pop     edi
        pop     esi
        ret     0
_memcpy_128 ENDP
```

Lors de la copie de 123 octets, 30 mots de 32-bit sont tout d'abord copiés en utilisant MOVSD (ce qui fait 120 octets), puis 2 octets sont copiés en utilisant MOVSW, puis un autre octet en utilisant MOVSB.

Listing 3.51 : MSVC 2010 avec optimisation

```
_inbuf$ = 8             ; size = 4
_outbuf$ = 12           ; size = 4
_memcpy_123 PROC
        push    esi
        mov     esi, DWORD PTR _inbuf$[esp]
        push    edi
        mov     edi, DWORD PTR _outbuf$[esp+4]
        add     edi, 10
        mov     ecx, 30
```

```
        rep movsd
        movsw
        movsb
        pop     edi
        pop     esi
        ret     0
_memcpy_123 ENDP
```

GCC utilise une grosse fonction universelle, qui fonctionne pour n'importe quelle taille de bloc:

Listing 3.52 : GCC 4.8.1 avec optimisation

```
memcpy_123:
.LFB3:
        push    edi
        mov     eax, 123
        push    esi
        mov     edx, DWORD PTR [esp+16]
        mov     esi, DWORD PTR [esp+12]
        lea     edi, [edx+10]
        test    edi, 1
        jne     .L24
        test    edi, 2
        jne     .L25
.L7:
        mov     ecx, eax
        xor     edx, edx
        shr     ecx, 2
        test    al, 2
        rep movsd
        je      .L8
        movzx   edx, WORD PTR [esi]
        mov     WORD PTR [edi], dx
        mov     edx, 2
.L8:
        test    al, 1
        je      .L5
        movzx   eax, BYTE PTR [esi+edx]
        mov     BYTE PTR [edi+edx], al
.L5:
        pop     esi
        pop     edi
        ret
.L24:
        movzx   eax, BYTE PTR [esi]
        lea     edi, [edx+11]
        add     esi, 1
        test    edi, 2
        mov     BYTE PTR [edx+10], al
        mov     eax, 122
        je      .L7
.L25:
        movzx   edx, WORD PTR [esi]
```

```
        add     edi, 2
        add     esi, 2
        sub     eax, 2
        mov     WORD PTR [edi-2], dx
        jmp     .L7
.LFE3:
```

Les fonctions de copie de mémoire universelles fonctionnent en général comme suit: calculer combien de mots de 32-bit peuvent être copiés, puis les copier en utilisant MOVSD, et enfin copier les octets restants.

Des fonctions de copie plus avancées et complexes utilisent les instructions SIMD et prennent aussi en compte l'alignement de la mémoire.

Voici un exemple de fonction strlen() SIMD: 1.36.2 on page 539.

memcmp()

Listing 3.53 : exemple memcmp()

```
int memcmp_1235(char *buf1, char *buf2)
{
        return memcmp(buf1, buf2, 1235);
};
```

Pour n'importe quelle taille de bloc, MSVC 2013 insère la même fonction universelle:

Listing 3.54 : avec optimisation MSVC 2010

```
_buf1$ = 8      ; size = 4
_buf2$ = 12     ; size = 4
_memcmp_1235 PROC
        mov     ecx, DWORD PTR _buf1$[esp-4]
        mov     edx, DWORD PTR _buf2$[esp-4]
        push    esi
        mov     esi, 1231
        npad    2
$LL5@memcmp_123:
        mov     eax, DWORD PTR [ecx]
        cmp     eax, DWORD PTR [edx]
        jne     SHORT $LN4@memcmp_123
        add     ecx, 4
        add     edx, 4
        sub     esi, 4
        jae     SHORT $LL5@memcmp_123
$LN4@memcmp_123:
        mov     al, BYTE PTR [ecx]
        cmp     al, BYTE PTR [edx]
        jne     SHORT $LN6@memcmp_123
        mov     al, BYTE PTR [ecx+1]
        cmp     al, BYTE PTR [edx+1]
        jne     SHORT $LN6@memcmp_123
        mov     al, BYTE PTR [ecx+2]
        cmp     al, BYTE PTR [edx+2]
```

```
        jne     SHORT $LN6@memcmp_123
        cmp     esi, -1
        je      SHORT $LN3@memcmp_123
        mov     al, BYTE PTR [ecx+3]
        cmp     al, BYTE PTR [edx+3]
        jne     SHORT $LN6@memcmp_123
$LN3@memcmp_123:
        xor     eax, eax
        pop     esi
        ret     0
$LN6@memcmp_123:
        sbb     eax, eax
        or      eax, 1
        pop     esi
        ret     0
_memcmp_1235 ENDP
```

strcat()

Ceci est un strcat() inline tel qu'il a été généré par MSVC 6.0. Il y a 3 parties visibles: 1) obtenir la longueur de la chaîne source (premier scasb) ; 2) obtenir la longueur de la chaîne destination (second scasb) ; 3) copier la chaîne source dans la fin de la chaîne de destination (paire movsd/movsb).

Listing 3.55 : strcat()

```
                lea     edi, [src]
                or      ecx, 0FFFFFFFFh
                repne scasb
                not     ecx
                sub     edi, ecx
                mov     esi, edi
                mov     edi, [dst]
                mov     edx, ecx
                or      ecx, 0FFFFFFFFh
                repne scasb
                mov     ecx, edx
                dec     edi
                shr     ecx, 2
                rep movsd
                mov     ecx, edx
                and     ecx, 3
                rep movsb
```

Script IDA

Il y a aussi un petit script IDA pour chercher et suivre de tels morceaux de code inline, que l'on rencontre fréquemment:

GitHub.

3.15 C99 restrict

Voici une raison pour laquelle les programmes en Fortran, dans certains cas, fonctionnent plus vite que ceux en C/C++.

```
void f1 (int* x, int* y, int* sum, int* product, int* sum_product, int* ⤦
    ↳ update_me, size_t s)
{
        for (int i=0; i<s; i++)
        {
                sum[i]=x[i]+y[i];
                product[i]=x[i]*y[i];
                update_me[i]=i*123; // some dummy value
                sum_product[i]=sum[i]+product[i];
        };
};
```

C'est un exemple très simple, qui contient une spécificité: le pointeur sur le tableau `update_me` peut-être un pointeur sur le tableau `sum`, le tableau `product` ou même le tableau `sum_product`—rien ne l'interdit, n'est-ce pas?

Le compilateur est parfaitement conscient de ceci, donc il génère du code avec quatre étapes dans le corps de la boucle:

- calcule le `sum[i]` suivant
- calcule le `product[i]` suivant
- calcule le `update_me[i]` suivant
- calcule le `sum_product[i]` suivant—à cette étape, nous devons charger depuis la mémoire les valeurs `sum[i]` et `product[i]` déjà calculées

Et-il possible d'optimiser la dernière étape? Puisque nous avons déjà calculé `sum[i]` et `product[i]`, il n'est pas nécessaire de les charger à nouveau depuis la mémoire.

Oui, mais le compilateurs n'est pas sûr que rien n'a été récris à la 3ème étape! Ceci est appelé «pointer aliasing », une situation dans laquelle le compilateur ne peut pas être sûr que la mémoire sur laquelle le pointeur pointe n'a pas été modifiée.

restrict dans le standard C99 [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007) 6.7.3/1] est une promesse faite par le programmeur au compilateur que les arguments de la fonction marqués par ce mot-clef vont toujours pointer vers des case mémoire différentes et ne vont jamais se recouper.

Pour être plus précis et décrire ceci formellement, *restrict* indique que seul ce pointeur est utilisé pour accéder un objet, et qu'aucun autre pointeur ne sera utilisé pour ceci.

On peut même dire que l'objet ne sera accéder que par un seul pointeur, si il est marqué comme *restrict*.

Ajoutons ce mot-clef à chaque argument pointeur:

```
void f2 (int* restrict x, int* restrict y, int* restrict sum, int* restrict⤦
    ↳  product, int* restrict sum_product,
```

```
        int* restrict update_me, size_t s)
{
        for (int i=0; i<s; i++)
        {
                sum[i]=x[i]+y[i];
                product[i]=x[i]*y[i];
                update_me[i]=i*123; // some dummy value
                sum_product[i]=sum[i]+product[i];
        };
};
```

Regardons le résultat:

Listing 3.56 : GCC x64: f1()

```
f1:
        push    r15 r14 r13 r12 rbp rdi rsi rbx
        mov     r13, QWORD PTR 120[rsp]
        mov     rbp, QWORD PTR 104[rsp]
        mov     r12, QWORD PTR 112[rsp]
        test    r13, r13
        je      .L1
        add     r13, 1
        xor     ebx, ebx
        mov     edi, 1
        xor     r11d, r11d
        jmp     .L4
.L6:
        mov     r11, rdi
        mov     rdi, rax
.L4:
        lea     rax, 0[0+r11*4]
        lea     r10, [rcx+rax]
        lea     r14, [rdx+rax]
        lea     rsi, [r8+rax]
        add     rax, r9
        mov     r15d, DWORD PTR [r10]
        add     r15d, DWORD PTR [r14]
        mov     DWORD PTR [rsi], r15d         ; stocker dans sum[]
        mov     r10d, DWORD PTR [r10]
        imul    r10d, DWORD PTR [r14]
        mov     DWORD PTR [rax], r10d         ; stocker dans product[]
        mov     DWORD PTR [r12+r11*4], ebx    ; stocker dans update_me[]
        add     ebx, 123
        mov     r10d, DWORD PTR [rsi]         ; recharger sum[i]
        add     r10d, DWORD PTR [rax]         ; recharger product[i]
        lea     rax, 1[rdi]
        cmp     rax, r13
        mov     DWORD PTR 0[rbp+r11*4], r10d  ; stocker dans sum_product[]
        jne     .L6
.L1:
        pop     rbx rsi rdi rbp r12 r13 r14 r15
        ret
```

Listing 3.57 : GCC x64: f2()

```
f2:
        push    r13 r12 rbp rdi rsi rbx
        mov     r13, QWORD PTR 104[rsp]
        mov     rbp, QWORD PTR 88[rsp]
        mov     r12, QWORD PTR 96[rsp]
        test    r13, r13
        je      .L7
        add     r13, 1
        xor     r10d, r10d
        mov     edi, 1
        xor     eax, eax
        jmp     .L10
.L11:
        mov     rax, rdi
        mov     rdi, r11
.L10:
        mov     esi, DWORD PTR [rcx+rax*4]
        mov     r11d, DWORD PTR [rdx+rax*4]
        mov     DWORD PTR [r12+rax*4], r10d  ; stocker dans update_me[]
        add     r10d, 123
        lea     ebx, [rsi+r11]
        imul    r11d, esi
        mov     DWORD PTR [r8+rax*4], ebx    ; stocker dans sum[]
        mov     DWORD PTR [r9+rax*4], r11d   ; stocker dans product[]
        add     r11d, ebx
        mov     DWORD PTR 0[rbp+rax*4], r11d ; stocker dans sum_product[]
        lea     r11, 1[rdi]
        cmp     r11, r13
        jne     .L11
.L7:
        pop     rbx rsi rdi rbp r12 r13
        ret
```

La différence entre les fonctions `f1()` et `f2()` compilées est la suivante: dans `f1()`, `sum[i]` et `product[i]` sont rechargés au milieu de la boucle, et il n'y a rien de tel dans `f2()`, les valeurs déjà calculées sont utilisées, puisque nous avons «promis» au compilateur que rien ni personne ne changera les valeurs pendant l'exécution du corps de la boucle, donc il est «certain» qu'il n'y a pas besoin de recharger la valeur depuis la mémoire.

Étonnamment, le second exemple est plus rapide.

Mais que se passe-t-il si les pointeurs dans les arguments de la fonction se modifient d'une manière ou d'une autre?

Ceci est du ressort de la conscience du programmeur, et le résultat sera incorrect.

Retournons au Fortran.

Les compilateurs de ce langage traitent tous les pointeurs de cette façon, donc lorsqu'il n'est pas possible d'utiliser *restrict* en C, Fortran peut générer du code plus rapide dans ces cas.

À quel point est-ce pratique?

Dans les cas où le fonction travaille avec des gros blocs en mémoire.

C'est le cas en algèbre linéaire, par exemple.

Les superordinateurs/HPC[15] utilisent beaucoup d'algèbre linéaire, c'est probablement pourquoi,
traditionnellement, Fortran y est encore utilisé [Eugene Loh, *The Ideal HPC Programming Language*, (2010)].

Mais lorsque le nombre d'itérations n'est pas très important, certainement, le gain en vitesse ne doit pas être significatif.

3.16 Fonction *abs()* sans branchement

Retravaillons un exemple que nous avons vu avant 1.18.2 on page 185 et demandons-nous, est-il possible de faire une version sans branchement de la fonction en code x86?

```
int my_abs (int i)
{
        if (i<0)
                return -i;
        else
                return i;
};
```

Et la réponse est oui.

3.16.1 GCC 4.9.1 x64 avec optimisation

Nous pouvons le voir si nous compilons en utilisant GCC 4.9 avec optimisation:

Listing 3.58 : GCC 4.9 x64 avec optimisation

```
my_abs:
        mov     edx, edi
        mov     eax, edi
        sar     edx, 31
; EDX contient ici 0xFFFFFFFF si le signe de la valeur en entrée est moins
; EDX contient 0 si le signe de la valeur en entrée est plus (0 inclus)
; les deux instructions suivantes ont un effet seulement si EDX contient
    0xFFFFFFFF
; et aucun si EDX contient 0
        xor     eax, edx
        sub     eax, edx
        ret
```

Voici comment ça fonctionne:

Décaler arithmétiquement la valeur en entrée par 31.

[15] High-Performance Computing

Le décalage arithmétique implique l'extension du signe, donc si le MSB est 1, les 32 bits seront tous remplis avec 1, ou avec 0 sinon.

Autrement dit, l'instruction `SAR REG, 31` donne `0xFFFFFFFF` si le signe était négatif et 0 s'il était positif.

Après l'exécution de SAR, nous avons cette valeur dans EDX.

Puis, si la valeur est `0xFFFFFFFF` (i.e., le signe est négatif), la valeur en entrée est inversée (car `XOR REG, 0xFFFFFFFF` est en effet une opération qui inverse tous les bits).

Puis, à nouveau, si la valeur est `0xFFFFFFFF` (i.e., le signe est négatif), 1 est ajouté au résultat final (car soustraire -1 d'une valeur revient à l'incrémenter).

Inverser tous les bits et incrémenter est exactement la façon dont une valeur en complément à deux est multipliée par -1.

Nous pouvons observer que les deux dernières instructions font quelque chose si le signe de la valeur en entrée est négatif.

Autrement (si le signe est positif) elles ne font rien du tout, laissant la valeur en entrée inchangée.

L'algorithme est expliqué dans [Henry S. Warren, *Hacker's Delight*, (2002)2-4].

Il est difficile de dire comment a fait GCC, l'a-t-il déduit lui-même ou un pattern correspondant parmi ceux connus?

3.16.2 GCC 4.9 ARM64 avec optimisation

GCC 4.9 pour ARM64 génère en gros la même chose, il décide juste d'utiliser des registres 64-bit complets.

Il y a moins d'instructions, car la valeur en entrée peut être décalée en utilisant un suffixe d'instruction («asr») au lieu d'une instruction séparée.

Listing 3.59 : GCC 4.9 ARM64 avec optimisation

```
my_abs:
; étendre le signe de la valeur 32-bit en entrée dans le registre X0 64-bit:
        sxtw    x0, w0
        eor     x1, x0, x0, asr 63
; X1=X0^(X0>>63) (le décalage est arithmétique)
        sub     x0, x1, x0, asr 63
; X0=X1-(X0>>63)=X0^(X0>>63)-(X0>>63)
; (tous les décalages sont arithmétiques)
        ret
```

3.17 Fonctions variadiques

Les fonctions comme `printf()` et `scanf()` peuvent avoir un nombre variable d'arguments. Comment sont-ils accédés?

3.17.1 Calcul de la moyenne arithmétique

Imaginons que nous voulions calculer la moyenne arithmétique, et que pour une raison quelconque, nous voulions passer toutes les valeurs comme arguments de la fonction.

Mais il est impossible d'obtenir le nombre d'arguments dans une fonction variadique en C/C++, donc indiquons la valeur -1 comme terminateur.

Utilisation de la macro va_arg

Il y a le fichier d'entête standard stdarg.h qui défini des macros pour prendre en compte de tels arguments.

Les fonctions `printf()` et `scanf()` l'utilisent aussi.

```
#include <stdio.h>
#include <stdarg.h>

int arith_mean(int v, ...)
{
        va_list args;
        int sum=v, count=1, i;
        va_start(args, v);

        while(1)
        {
                i=va_arg(args, int);
                if (i==-1) // terminateur
                        break;
                sum=sum+i;
                count++;
        }

        va_end(args);
        return sum/count;
};

int main()
{
        printf ("%d\n", arith_mean (1, 2, 7, 10, 15, -1 /* terminateur */))↙
    ↪ ;
};
```

Le premier argument doit être traité comme un argument normal.

Tous les autres arguments sont chargés en utilisant la macro va_arg et ensuite ajoutés.

Qu'y a-t-il à l'intérieur?

Convention d'appel *cdecl*

Listing 3.60 : MSVC 6.0 avec optimisation

```
_v$ = 8
_arith_mean PROC NEAR
        mov     eax, DWORD PTR _v$[esp-4] ; charger le 1er argument dans sum
        push    esi
        mov     esi, 1                    ; count=1
        lea     edx, DWORD PTR _v$[esp]   ; adresse du 1er argument
$L838:
        mov     ecx, DWORD PTR [edx+4]    ; charger l'argument suivant
        add     edx, 4                    ; décaler le pointeur sur
    l'argument suivant
        cmp     ecx, -1                   ; est-ce -1?
        je      SHORT $L856               ; sortir si oui
        add     eax, ecx                  ; sum = sum + argument chargé
        inc     esi                       ; count++
        jmp     SHORT $L838
$L856:
; calculer le quotient

        cdq
        idiv    esi
        pop     esi
        ret     0
_arith_mean ENDP

$SG851  DB      '%d', 0aH, 00H

_main   PROC NEAR
        push    -1
        push    15
        push    10
        push    7
        push    2
        push    1
        call    _arith_mean
        push    eax
        push    OFFSET FLAT:$SG851 ; '%d'
        call    _printf
        add     esp, 32
        ret     0
_main   ENDP
```

Les arguments, comme on le voit, sont passés à `main()` un par un.

Le premier argument est poussé sur la pile locale en premier.

La valeur terminale (-1) est poussée plus tard.

La fonction `arith_mean()` prend la valeur du premier argument et le stocke dans la variable sum.

Puis, elle met dans le registre EDX l'adresse du second argument, prend sa valeur, l'ajoute à sum, et fait cela dans une boucle infinie, jusqu'à ce que -1 soit trouvé.

Lorsqu'il est rencontré, la somme est divisée par le nombre de valeurs (en excluant -1) et le quotient est renvoyé.

Donc, autrement dit, la fonction traite le morceau de pile comme un tableau de valeurs entières d'une longueur infinie.

Maintenant nous pouvons comprendre pourquoi la convention d'appel *cdecl* nous force à pousser le premier argument au moins sur la pile.

Car sinon, il ne serait pas possible de trouver le premier argument, ou, pour les fonctions du genre de printf, il ne serait pas possible de trouver l'adresse de la chaîne de format.

Convention d'appel basée sur les registres

Le lecteur attentif pourrait demander, qu'en est-il de la convention d'appel où les tous premiers arguments sont passés dans des registres? Regardons:

Listing 3.61 : MSVC 2012 x64 avec optimisation

```
$SG3013 DB      '%d', 0aH, 00H

v$ = 8
arith_mean PROC
        mov     DWORD PTR [rsp+8], ecx   ; 1er argument
        mov     QWORD PTR [rsp+16], rdx  ; 2nd argument
        mov     QWORD PTR [rsp+24], r8   ; 3ème argument
        mov     eax, ecx                 ; sum = 1er argument
        lea     rcx, QWORD PTR v$[rsp+8] ; pointeur sur le 2nd argument
        mov     QWORD PTR [rsp+32], r9   ; 4ème argument
        mov     edx, DWORD PTR [rcx]     ; charger le 2nd argument
        mov     r8d, 1                   ; count=1
        cmp     edx, -1                  ; est-ce que le 2nd argument est -1?
        je      SHORT $LN8@arith_mean    ; sortir si oui
$LL3@arith_mean:
        add     eax, edx                 ; sum = sum + argument chargé
        mov     edx, DWORD PTR [rcx+8]   ; charger l'argument suivant
        lea     rcx, QWORD PTR [rcx+8]   ; décaler le pointeur pour pointer
                                         ; sur l'argument après le suivant
        inc     r8d                      ; count++
        cmp     edx, -1                  ; est-ce que l'argument chargé est
    -1?
        jne     SHORT $LL3@arith_mean    ; aller au début de la boucle si non
$LN8@arith_mean:
; calculer le quotient
        cdq
        idiv    r8d
        ret     0
arith_mean ENDP

main    PROC
        sub     rsp, 56
        mov     edx, 2
        mov     DWORD PTR [rsp+40], -1
```

```
        mov     DWORD PTR [rsp+32], 15
        lea     r9d, QWORD PTR [rdx+8]
        lea     r8d, QWORD PTR [rdx+5]
        lea     ecx, QWORD PTR [rdx-1]
        call    arith_mean
        lea     rcx, OFFSET FLAT:$SG3013
        mov     edx, eax
        call    printf
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
```

Nous voyons que les 4 premiers arguments sont passés dans des registres, et les deux autres—par la pile.

La fonction `arith_mean()` place d'abord ces 4 arguments dans le *Shadow Space* puis traite le *Shadow Space* et la pile derrière comme s'il s'agissait d'un tableau continu!

Qu'en est-il de GCC? Les choses sont légèrement plus maladroites ici, car maintenant la fonction est divisée en deux parties: la première partie sauve les registres dans la «zone rouge», traite cet espace, et la seconde partie traite la pile:

Listing 3.62 : GCC 4.9.1 x64 avec optimisation

```
arith_mean:
        lea     rax, [rsp+8]
        ; sauver les 6 registrers en entrée dans
        ; la red zone sur la pile locale
        mov     QWORD PTR [rsp-40], rsi
        mov     QWORD PTR [rsp-32], rdx
        mov     QWORD PTR [rsp-16], r8
        mov     QWORD PTR [rsp-24], rcx
        mov     esi, 8
        mov     QWORD PTR [rsp-64], rax
        lea     rax, [rsp-48]
        mov     QWORD PTR [rsp-8], r9
        mov     DWORD PTR [rsp-72], 8
        lea     rdx, [rsp+8]
        mov     r8d, 1
        mov     QWORD PTR [rsp-56], rax
        jmp     .L5
.L7:
        ; traiter les arguments sauvés
        lea     rax, [rsp-48]
        mov     ecx, esi
        add     esi, 8
        add     rcx, rax
        mov     ecx, DWORD PTR [rcx]
        cmp     ecx, -1
        je      .L4
.L8:
        add     edi, ecx
        add     r8d, 1
```

```
.L5:
        ; décider, quelle partie traiter maintenant.
        ; est-ce que le nombre d'arguments actuel est inférieur ou égal à 6?
        cmp     esi, 47
        jbe     .L7             ; non, traiter les arguments sauvegardés;
        ; traiter les arguments de la pile
        mov     rcx, rdx
        add     rdx, 8
        mov     ecx, DWORD PTR [rcx]
        cmp     ecx, -1
        jne     .L8
.L4:
        mov     eax, edi
        cdq
        idiv    r8d
        ret

.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        mov     edx, 7
        mov     esi, 2
        mov     edi, 1
        mov     r9d, -1
        mov     r8d, 15
        mov     ecx, 10
        xor     eax, eax
        call    arith_mean
        mov     esi, OFFSET FLAT:.LC1
        mov     edx, eax
        mov     edi, 1
        xor     eax, eax
        add     rsp, 8
        jmp     __printf_chk
```

À propos, un usage similaire du *Shadow Space* est aussi considéré ici: 6.1.8 on page 962.

Utilisation du pointeur sur le premier argument de la fonction

L'exemple peut être récrit sans la macro va_arg :

```
#include <stdio.h>

int arith_mean(int v, ...)
{
        int *i=&v;
        int sum=*i, count=1;
        i++;

        while(1)
        {
```

```
                if ((*i)==-1) // terminator
                        break;
                sum=sum+(*i);
                count++;
                i++;
        }

        return sum/count;
};

int main()
{
        printf ("%d\n", arith_mean (1, 2, 7, 10, 15, -1 /* terminator */));
        // test: https://www.wolframalpha.com/input/?i=mean(1,2,7,10,15)
};
```

Autrement dit, si l'argument mis est un tableau de mots (32-bit ou 64-bit), nous devons juste énumérer les éléments du tableau en commençant par le premier.

3.17.2 Cas de la fonction *vprintf()*

De nombreux programmeurs définissent leur propre fonction de logging, qui prend une chaîne de format du type de celle de printf+une liste variable d'arguments.

Un autre exemple répandu est la fonction die(), qui affiche un message et sort.

Nous avons besoin d'un moyen de transmettre un nombre d'arguments inconnu et de les passer à la fonction `printf()`. Mais comment?

À l'aide des fonctions avec un «v» dans le nom.

Une d'entre elles est *vprintf()* : elle prend une chaîne de format et un pointeur sur une variable du type `va_list` :

```
#include <stdlib.h>
#include <stdarg.h>

void die (const char * fmt, ...)
{
        va_list va;
        va_start (va, fmt);

        vprintf (fmt, va);
        exit(0);
};
```

En examinant plus précisément, nous voyons que `va_list` est un pointeur sur un tableau. Compilons:

Listing 3.63 : MSVC 2010 avec optimisation

```
_fmt$ = 8
_die    PROC
        ; charger le 1er argument (format-string)
```

```
        mov     ecx, DWORD PTR _fmt$[esp-4]
        ; obtenir un pointeur sur le 2nd argument
        lea     eax, DWORD PTR _fmt$[esp]
        push    eax             ; passer un pointeur
        push    ecx
        call    _vprintf
        add     esp, 8
        push    0
        call    _exit
$LN3@die:
        int     3
_die    ENDP
```

Nous voyons que tout ce que fait notre fonction est de prendre un pointeur sur les arguments et le passe à la fonction *vprintf()*, et que cette fonction le traite comme un tableau infini d'arguments!

Listing 3.64 : MSVC 2012 x64 avec optimisation

```
fmt$ = 48
die     PROC
        ; sauver les 4 premiers arguments dans le Shadow Space
        mov     QWORD PTR [rsp+8], rcx
        mov     QWORD PTR [rsp+16], rdx
        mov     QWORD PTR [rsp+24], r8
        mov     QWORD PTR [rsp+32], r9
        sub     rsp, 40
        lea     rdx, QWORD PTR fmt$[rsp+8] ; passer un pointeur sur le 1er
    argument
        ; ici RCX pointe toujours sur le 1er argument (format-string) de
    die()
        ; donc vprintf() va prendre son argument dans RCX
        call    vprintf
        xor     ecx, ecx
        call    exit
        int     3
die     ENDP
```

3.17.3 Cas Pin

Il est intéressant de noter que certaines fonctions du framework DBI[16] Pin prennent un nombre variable d'arguments:

```
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
                IARG_INST_PTR,
                IARG_MEMORYOP_EA, memOp,
                IARG_END);
```

(pinatrace.cpp)

Et voici comment la fonction `INS_InsertPredicatedCall()` est déclarée:

[16] Dynamic Binary Instrumentation

```
extern VOID INS_InsertPredicatedCall(INS ins, IPOINT ipoint, AFUNPTR funptr↙
    ↳ , ...);
```

(pin_client.PH)

Ainsi, les constantes avec un nom débutant par IARG_ sont des sortes d'arguments pour la fonction, qui sont manipulés à l'intérieur de INS_InsertPredicatedCall(). Vous pouvez passer autant d'arguments que vous en avez besoin. Certaines commandes ont des arguments additionnels, d'autres non. Liste complète des arguments: https://software.intel.com/sites/landingpage/pintool/docs/58423/Pin/html/group__INST__ARGS.html. Et il faut un moyen pour détecter la fin de la liste des arguments, donc la liste doit être terminée par la constante IARG_END, sans laquelle, la fonction essayerait de traiter les données indéterminées dans la pile locale comme des arguments additionnels.

Aussi, dans [Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)] nous pouvons trouver un bel exemple de routines C/C++ très similaires à *pack/unpack*[17] en Python.

3.17.4 Exploitation de chaîne de format

Il y a une erreur courante, celle d'écrire printf(string) au lieu de puts(string) ou printf("%s", string). Si l'attaquant peut mettre son propre texte dans string, il peut planter le processus ou accéder aux variables de la pile locale.

Regardons ceci:

```
#include <stdio.h>

int main()
{
        char *s1="hello";
        char *s2="world";
        char buf[128];

        // do something mundane here
        strcpy (buf, s1);
        strcpy (buf, " ");
        strcpy (buf, s2);

        printf ("%s");
};
```

Veuillez noter que printf() n'a pas d'argument supplémentaire autre que la chaîne de format.

Maintenant, imaginons que c'est l'attaquant qui a mis la chaîne %s dans le premier argument du dernier printf(). Je compile cet exemple en utilisant GCC 5.4.0 sous Ubuntu x86, et l'exécutable résultant affiche la chaîne «world» s'il est exécuté!

[17] https://docs.python.org/3/library/struct.html

Si je compile avec l'optimisation, `printf()` affiche n'importe quoi, aussi—probablement, l'appel à strcpy() a été optimisé et/ou les variables locales également. De même, le résultat pour du code x64 sera différent, pour différents compilateurs, OS, etc.

Maintenant, disons que l'attaquant peut passer la chaîne suivante à l'appel de `printf()` : `%x %x %x %x %x`. Dans mon cas, la sortie est: « 80485c6 b7751b48 1 0 80485c0 » (ce sont simplement des valeurs de la pile locale). Vous voyez, il y a les valeurs 1 et 0, et des pointeurs (le premier est probablement un pointeur sur la chaîne « world »). Donc si l'attaquant passe la chaîne `%s %s %s %s %s`, le processus va se planter, car `printf()` traite 1 et/ou 0 comme des pointeurs sur une chaîne, essaye de lire des caractères et échoue.

Encore pire, il pourrait y avoir `sprintf (buf, string)` dans le code, où `buf` est un buffer dans la pile locale avec un taille de 1024 octets ou autre, l'attaquant pourrait préparer une chaîne de telle sorte que `buf` serait débordé, peut-être même de façon à conduire à l'exécution de code.

De nombreux logiciels bien connus et très utilisés étaient (ou sont encore) vulnérables:

> QuakeWorld went up, got to around 4000 users, then the master server exploded.
> (QuakeWorld est arrivé, monté à environ 4000 utilisateurs, puis le serveur master a explosé.)
> Disrupter and cohorts are working on more robust code now.
> (Les perturbateurs et cohortes travaillent maintenant sur un code plus robuste.)
> If anyone did it on purpose, how about letting us know... (It wasn't all the people that tried %s as a name)
> (Si quelqu'un l'a fait exprès, pourquoi ne pas nous le faire savoir... (Ce n'est pas tout le monde qui a essayé %s comme nom))

(John Carmack's .plan file, 17-Dec-1996[18])

De nos jours, tous les compilateurs dignes de ce nom avertissent à propos de ceci.

Un autre problème qui est moins connu, c'est l'argument `%n` de `printf()` : lorsque `printf()` l'atteint dans la chaîne de format, il écrit le nombre de caractères écrits jusqu'ici dans l'argument correspondant: stackoverflow.com. Ainsi, un attaquant peut zapper les variables locales en passant plusieurs commandes `%n` dans la chaîne de format.

3.18 Ajustement de chaînes

Un traitement de chaîne très courant est la suppression de certains caractères au début et/où à la fin.

[18] https://github.com/ESWAT/john-carmack-plan-archive/blob/33ae52fdba46aa0d1abfed6fc7598233748541c0/by_day/johnc_plan_19961217.txt

Dans cet exemple, nous allons travailler avec une fonction qui supprime tous les caractères newline (CR[19]/LF[20]) à la fin de la chaîne entrée:

```
#include <stdio.h>
#include <string.h>

char* str_trim (char *s)
{
        char c;
        size_t str_len;

        // fonctionne tant que \r ou \n se trouve en fin de la chaîne
        // s'arrête si un autre caractère s'y trouve ou si la chaîne est vide
        // (au début ou au cours de notre opération)
        for (str_len=strlen(s); str_len>0 && (c=s[str_len-1]); str_len--)
        {
                if (c=='\r' || c=='\n')
                        s[str_len-1]=0;
                else
                        break;
        };
        return s;
};

int main()
{
        // test

        // strdup() est utilisé pour copier du texte de chaîne dans le
    segment de données,
        // car autrement ça va crasher sur Linux,
        // où les chaîne de texte sont allouées dans le segment de données
    constantes,
        // et n'est pas modifiable.

        printf ("[%s]\n", str_trim (strdup("")));
        printf ("[%s]\n", str_trim (strdup("\n")));
        printf ("[%s]\n", str_trim (strdup("\r")));
        printf ("[%s]\n", str_trim (strdup("\n\r")));
        printf ("[%s]\n", str_trim (strdup("\r\n")));
        printf ("[%s]\n", str_trim (strdup("test1\r\n")));
        printf ("[%s]\n", str_trim (strdup("test2\n\r")));
        printf ("[%s]\n", str_trim (strdup("test3\n\r\n\r")));
        printf ("[%s]\n", str_trim (strdup("test4\n")));
        printf ("[%s]\n", str_trim (strdup("test5\r")));
        printf ("[%s]\n", str_trim (strdup("test6\r\r\r")));
};
```

L'argument en entrée est toujours renvoyé en sortie, ceci est pratique lorsque vous voulez chaîner les fonctions de traitement de chaîne, comme c'est fait ici dans la fonction `main()`.

[19]Carriage return (13 ou '\r' en C/C++)
[20]Line feed (10 ou '\n' en C/C++)

La seconde partie de for() (`str_len>0 && (c=s[str_len-1])`) est appelé le «short-circuit» en C/C++ et est très pratique [Dennis Yurichev, *C/C++ programming language notes*1.3.8].

Les compilateurs C/C++ garantissent une séquence d'évaluation de gauche à droite.

Donc, si la première clause est fausse après l'évaluation, la seconde n'est pas évaluée.

3.18.1 x64: MSVC 2013 avec optimisation

Listing 3.65 : MSVC 2013 x64 avec optimisation

```
s:$ = 8
str_trim PROC

; RCX est le premier argument de la fonction et il contient toujours un
    pointeur sur la chaîne
        mov     rdx, rcx
; ceci est la fonction strlen() inlined juste ici:
; mettre RAX à 0xFFFFFFFFFFFFFFFF (-1)
        or      rax, -1
$LL14@str_trim:
        inc     rax
        cmp     BYTE PTR [rcx+rax], 0
        jne     SHORT $LL14@str_trim
; est-ce que la chaîne en entrée est de longueur zéro? alors sortir:
        test    rax, rax
        je      SHORT $LN15@str_trim
; RAX contient la longueur de la chaîne
        dec     rcx
; RCX = s-1
        mov     r8d, 1
        add     rcx, rax
; RCX = s-1+strlen(s), i.e., ceci est l'adresse du dernier caractère de la
    chaîne
        sub     r8, rdx
; R8 = 1-s
$LL6@str_trim:
; charger le dernier caractère de la chaîne:
; sauter si son code est 13 ou 10:
        movzx   eax, BYTE PTR [rcx]
        cmp     al, 13
        je      SHORT $LN2@str_trim
        cmp     al, 10
        jne     SHORT $LN15@str_trim
$LN2@str_trim:
; le dernier caractère a un code de 13 ou 10
; écrire zéro à cet endroit:
        mov     BYTE PTR [rcx], 0
; décrémenter l'adresse du dernier caractère,
; donc il pointera sur le caractère précédent celui qui vient d'être effacé:
        dec     rcx
        lea     rax, QWORD PTR [r8+rcx]
```

```
; RAX = 1 - s + adresse du dernier caractère courant
; ainsi nous pouvons déterminer si nous avons atteint le premier caractère et
; nous devons arrêter, si c'est le cas
        test    rax, rax
        jne     SHORT $LL6@str_trim
$LN15@str_trim:
        mov     rax, rdx
        ret     0
str_trim ENDP
```

Tout d'abord, MSVC a inliné le code la fonction `strlen()`, car il en a conclus que ceci était plus rapide que le `strlen()` habituel + le coût de l'appel et du retour. Ceci est appelé de l'inlining: 3.14 on page 649.

La première instruction de `strlen()` mis en ligne est
`OR RAX, 0xFFFFFFFFFFFFFFFF`.

MSVC utilise souvent `OR` au lieu de `MOV RAX, 0xFFFFFFFFFFFFFFFF`, car l'opcode résultant est plus court.

Et bien sûr, c'est équivalent: tous les bits sont mis à 1, et un nombre avec tous les bits mis vaut -1 en complément à 2.

On peut se demander pourquoi le nombre -1 est utilisé dans `strlen()`. À des fins d'optimisation, bien sûr. Voici le code que MSVC a généré:

Listing 3.66 : Inlined `strlen()` by MSVC 2013 x64

```
; RCX = pointeur sur la chaîne en entrée
; RAX = longueur actuelle de la chaîne
        or      rax, -1
label:
        inc     rax
        cmp     BYTE PTR [rcx+rax], 0
        jne     SHORT label
; RAX = longueur de la chaîne
```

Essayez d'écrite plus court si vous voulez initialiser le compteur à 0! OK, essayons:

Listing 3.67 : Our version of `strlen()`

```
; RCX = pointeur sur la chaîne en entrée
; RAX = longueur actuelle de la chaîne
        xor     rax, rax
label:
        cmp     byte ptr [rcx+rax], 0
        jz      exit
        inc     rax
        jmp     label
exit:
; RAX = longueur de la chaîne
```

Nous avons échoué. Nous devons utilisé une instruction JMP additionnelle!

Donc, ce que le compilateur de MSVC 2013 a fait, c'est de déplacer l'instruction INC avant le chargement du caractère courant.

Si le premier caractère est 0, c'est OK, RAX contient 0 à ce moment, donc la longueur de la chaîne est 0.

Le reste de cette fonction semble facile à comprendre.

3.18.2 x64: GCC 4.9.1 sans optimisation

```
str_trim:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     QWORD PTR [rbp-24], rdi
; la première partie de for() commence ici
        mov     rax, QWORD PTR [rbp-24]
        mov     rdi, rax
        call    strlen
        mov     QWORD PTR [rbp-8], rax   ; str_len
; la première partie de for() se termine ici
        jmp     .L2
; le corps de for() commence ici
.L5:
        cmp     BYTE PTR [rbp-9], 13     ; c=='\r'?
        je      .L3
        cmp     BYTE PTR [rbp-9], 10     ; c=='\n'?
        jne     .L4
.L3:
        mov     rax, QWORD PTR [rbp-8]   ; str_len
        lea     rdx, [rax-1]             ; EDX=str_len-1
        mov     rax, QWORD PTR [rbp-24]  ; s
        add     rax, rdx                 ; RAX=s+str_len-1
        mov     BYTE PTR [rax], 0        ; s[str_len-1]=0
; le corps de for() se termine ici
; la troisième partie de for() commence ici
        sub     QWORD PTR [rbp-8], 1     ; str_len--
; la troisième partie de for() se termine ici
.L2:
; la deuxième partie de for() commence ici
        cmp     QWORD PTR [rbp-8], 0     ; str_len==0?
        je      .L4                      ; alors sortir
; tester la seconde clause, et charger "c"
        mov     rax, QWORD PTR [rbp-8]   ; RAX=str_len
        lea     rdx, [rax-1]             ; RDX=str_len-1
        mov     rax, QWORD PTR [rbp-24]  ; RAX=s
        add     rax, rdx                 ; RAX=s+str_len-1
        movzx   eax, BYTE PTR [rax]  ; AL=s[str_len-1]
        mov     BYTE PTR [rbp-9], al     ; stocker l caractère chargé dans
    "c"
        cmp     BYTE PTR [rbp-9], 0      ; est-ce zéro?
        jne     .L5                      ; oui? alors sortir
; la deuxième partie de for() se termine ici
.L4:
```

```
; renvoyer "s"
        mov     rax, QWORD PTR [rbp-24]
        leave
        ret
```

Les commentaires ont été ajoutés par l'auteur du livre.

Après l'exécution de `strlen()`, le contrôle est passé au label L2, et ici deux clauses sont vérifiées, l'une après l'autre.

La seconde ne sera jamais vérifiée, si la première (*str_len==0*) est fausse (ceci est un «short-circuit» (court-circuit)).

Maintenant regardons la forme courte de cette fonction:

- Première partie de for() (appel à `strlen()`)
- goto L2
- L5: corps de for(). sauter à la fin, si besoin
- troisième partie de for() (décrémenter str_len)
- L2: deuxième partie de for() : vérifier la première clause, puis la seconde. sauter au début du corps de la boucle ou sortir.
- L4: // sortir
- renvoyer s

3.18.3 x64: GCC 4.9.1 avec optimisation

```
str_trim:
        push    rbx
        mov     rbx, rdi
; RBX sera toujours "s"
        call    strlen
; tester si str_len==0 et sortir si c'est la cas
        test    rax, rax
        je      .L9
        lea     rdx, [rax-1]
; RDX contiendra toujours la valeur str_len-1, pas str_len
; donc RDX est plutôt comme une variable sur l'index du buffer
        lea     rsi, [rbx+rdx]      ; RSI=s+str_len-1
        movzx   ecx, BYTE PTR [rsi] ; charger le caractère
        test    cl, cl
        je      .L9                 ; sortir si c'est zéro
        cmp     cl, 10
        je      .L4
        cmp     cl, 13              ; sortir si ce n'est ni '\n' ni '\r'
        jne     .L9
.L4:
; ceci est une instruction bizarre, nous voulons RSI=s-1 ici.
; c'est possible de l'obtenir avec MOV RSI, EBX / DEC RSI
; mais ce sont deux instructions au lieu d'une
        sub     rsi, rax
```

```
; RSI = s+str_len-1-str_len = s-1
; la boucle principale commence
.L12:
        test    rdx, rdx
; stocker zéro à l'adresse s-1+str_len-1+1 = s-1+str_len = s+str_len-1
        mov     BYTE PTR [rsi+1+rdx], 0
; tester si str_len-1==0. sortir si oui.
        je      .L9
        sub     rdx, 1                  ; équivalent à str_len--
; charger le caractère suivant à l'adresse s+str_len-1
        movzx   ecx, BYTE PTR [rbx+rdx]
        test    cl, cl                  ; est-ce zéro? sortir si oui
        je      .L9
        cmp     cl, 10                  ; est-ce '\n'?
        je      .L12
        cmp     cl, 13                  ; est-ce '\r'?
        je      .L12
.L9:
; renvoyer "s"
        mov     rax, rbx
        pop     rbx
        ret
```

Maintenant, c'est plus complexe.

Le code avant le début du corps de la boucle est exécuté une seule fois, mais il contient le test des caractères CR/LF aussi! À quoi sert cette duplication du code?

La façon courante d'implémenter la boucle principale est sans doute ceci:

- (début de la boucle) tester la présence des caractères CR/LF, décider
- stocker le caractère zéro

Mais GCC a décidé d'inverser ces deux étapes.

Bien sûr, *stocker le caractère zéro* ne peut pas être la première étape, donc un autre test est nécessaire:

- traiter le premier caractère. matcher avec CR/LF, sortir si le caractère n'est pas CR/LF
- (début de la boucle) stocker le caractère zéro
- tester la présence des caractères CR/LF, décider

Maintenant la boucle principale est très courte, ce qui est bon pour les derniers CPUs.

Le code n'utilise pas la variable str_len, mais str_len-1. Donc c'est plus comme un index dans un buffer.

Apparemment, GCC a remarqué que l'expression str_len-1 est utilisée deux fois.

Donc, c'est mieux d'allouer une variable qui contient toujours une valeur qui est plus petite que la longueur actuelle de la chaîne de un, et la décrémente (ceci a le même effet que de décrémenter la variable str_len).

3.18.4 ARM64: GCC (Linaro) 4.9 sans optimisation

Cette implémentation est simple:

Listing 3.68 : GCC (Linaro) 4.9 sans optimisation

```
str_trim:
        stp     x29, x30, [sp, -48]!
        add     x29, sp, 0
        str     x0, [x29,24] ; copier l'argument en entrée dans la pile
    locale
        ldr     x0, [x29,24] ; s
        bl      strlen
        str     x0, [x29,40] ; la variable str_len est dans la pile locale
        b       .L2
; la boucle principale commence
.L5:
        ldrb    w0, [x29,39]
; W0=c
        cmp     w0, 13       ; est-ce '\r'?
        beq     .L3
        ldrb    w0, [x29,39]
; W0=c
        cmp     w0, 10       ; est-ce '\n'?
        bne     .L4          ; sauter à la sortie si non
.L3:
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        ldr     x1, [x29,24]
; X1=s
        add     x0, x1, x0
; X0=s+str_len-1
        strb    wzr, [x0]    ; écrire l'octet à s+str_len-1
; décrémenter str_len:
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        str     x0, [x29,40]
; sauver X0 (or str_len-1) dans la pile locale
.L2:
        ldr     x0, [x29,40]
; str_len==0?
        cmp     x0, xzr
; sauter alors à la sortie
        beq     .L4
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        ldr     x1, [x29,24]
; X1=s
        add     x0, x1, x0
```

```
; X0=s+str_len-1
; charger l'octet à l'adresse s+str_len-1 dans W0
        ldrb    w0, [x0]
        strb    w0, [x29,39] ; stocker l'octet chargé dans "c"
        ldrb    w0, [x29,39] ; le recharger
; est-ce l'octet zéro?
        cmp     w0, wzr
; sauter à la sortie, si c'est zéro ou en L5 sinon
        bne     .L5
.L4:
; renvoyer s
        ldr     x0, [x29,24]
        ldp     x29, x30, [sp], 48
        ret
```

3.18.5 ARM64: GCC (Linaro) 4.9 avec optimisation

Ceci est une optimisation plus avancée.

Le premier caractère est chargé au début, et comparé avec 10 (le caractère LF).

Les caractères sont ensuite chargés dans la boucle principale, pour les caractères après le premier.

Ceci est quelque peu similaire à l'exemple 3.18.3 on page 679.

Listing 3.69 : GCC (Linaro) 4.9 avec optimisation

```
str_trim:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        str     x19, [sp,16]
        mov     x19, x0
; X19 contiendra toujours la valeur de "s"
        bl      strlen
; X0=str_len
        cbz     x0, .L9         ; sauter en L9 (sortir) si str_len==0
        sub     x1, x0, #1
; X1=X0-1=str_len-1
        add     x3, x19, x1
; X3=X19+X1=s+str_len-1
        ldrb    w2, [x19,x1]    ; charger l'octet à l'adresse
    X19+X1=s+str_len-1
; W2=octet chargé
        cbz     w2, .L9         ; est-ce zéro? sauter alors à la sortie
        cmp     w2, 10          ; est-ce '\n'?
        bne     .L15
.L12:
; corps de la boucle principale. Le caractère chargé est toujours 10 ou 13 à
    ce moment!
        sub     x2, x1, x0
; X2=X1-X0=str_len-1-str_len=-1
        add     x2, x3, x2
; X2=X3+X2=s+str_len-1+(-1)=s+str_len-2
```

```
        strb    wzr, [x2,1]     ; stocker l'octet zéro à l'adresse
    s+str_len-2+1=s+str_len-1
        cbz     x1, .L9         ; str_len-1==0? sauter à la sortie si oui
        sub     x1, x1, #1      ; str_len--
        ldrb    w2, [x19,x1]    ; charger le caractère suivant à l'adresse
    X19+X1=s+str_len-1
        cmp     w2, 10          ; est-ce '\n'?
        cbz     w2, .L9         ; sauter à la sortie, si c'est zéro
        beq     .L12            ;
    sauter au début du corps de la boucle, si c'est '\n'
.L15:
        cmp     w2, 13          ; est-ce '\r'?
        beq     .L12            ; oui, sauter au début du corps de la boucle
.L9:
; renvoyer "s"
        mov     x0, x19
        ldr     x19, [sp,16]
        ldp     x29, x30, [sp], 32
        ret
```

3.18.6 ARM: avec optimisation Keil 6/2013 (Mode ARM)

À nouveau, le compilateur tire partie des instructions conditionnelles du mode ARM, donc le code est bien plus compact.

Listing 3.70 : avec optimisation Keil 6/2013 (Mode ARM)

```
str_trim PROC
        PUSH     {r4,lr}
; R0=s
        MOV      r4,r0
; R4=s
        BL       strlen       ; strlen() prend la valeur de "s" dans R0
; R0=str_len
        MOV      r3,#0
; R3 contiendra toujours 0
|L0.16|
        CMP      r0,#0        ; str_len==0?
        ADDNE    r2,r4,r0     ; (si str_len!=0) R2=R4+R0=s+str_len
        LDRBNE   r1,[r2,#-1]  ; (si str_len!=0) R1=charger l'octet à
    l'adresse R2-1=s+str_len-1
        CMPNE    r1,#0        ; (si str_len!=0) comparer l'octet chargé avec
    0
        BEQ      |L0.56|      ; sauter à la sortie si str_len==0 ou si
    l'octet chargé est 0
        CMP      r1,#0xd      ; est-ce que l'octet chargé est '\r'?
        CMPNE    r1,#0xa      ;
    (si l'octet chargé n'est pas '\r') est-ce '\r'?
        SUBEQ    r0,r0,#1     ;
    (si l'octet chargé  est '\r' ou '\n') R0-- ou str_len--
        STRBEQ   r3,[r2,#-1]  ; (si l'octet chargé est '\r' ou '\n') stocker
    R3 (zéero) à l'adresse R2-1=s+str_len-1
        BEQ      |L0.16|      ;
    sauter au début de a boucle si l'octet chargé était '\r' ou '\n'
|L0.56|
```

```
; renvoyer "s"
        MOV      r0,r4
        POP      {r4,pc}
        ENDP
```

3.18.7 ARM: avec optimisation Keil 6/2013 (Mode Thumb)

Il y a moins d'instructions conditionnelles en mode Thumb, donc le code est plus simple.

Mais il y a des choses vraiment étranges aux offsets 0x20 et 0x1F (lignes 22 et 23). Pourquoi diable le compilateur Keil a-t-il fait ça? Honnêtement, c'est difficile de le dire.

Ça doit être une bizarrerie du processus d'optimisation de Keil. Néanmoins, le code fonctionne correctement

Listing 3.71 : avec optimisation Keil 6/2013 (Mode Thumb)

```
1  str_trim PROC
2          PUSH     {r4,lr}
3          MOVS     r4,r0
4  ; R4=s
5          BL       strlen         ; strlen() prend la valeur de "s" dans R0
6  ; R0=str_len
7          MOVS     r3,#0
8  ; R3 contiendra toujours 0
9          B        |L0.24|
10 |L0.12|
11         CMP      r1,#0xd        ; est-ce que l'octet chargé est '\r'?
12         BEQ      |L0.20|
13         CMP      r1,#0xa        ; est-ce que l'octet chargé est '\n'?
14         BNE      |L0.38|        ; sauter à la sortie si non
15 |L0.20|
16         SUBS     r0,r0,#1       ; R0-- ou str_len--
17         STRB     r3,[r2,#0x1f] ; stocker 0 à l'adresse R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1
18 |L0.24|
19         CMP      r0,#0          ; str_len==0?
20         BEQ      |L0.38|        ; oui? sauter à la sortie
21         ADDS     r2,r4,r0       ; R2=R4+R0=s+str_len
22         SUBS     r2,r2,#0x20    ; R2=R2-0x20=s+str_len-0x20
23         LDRB     r1,[r2,#0x1f] ; charger l'octet à l'adresse R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1 dans R1
24         CMP      r1,#0          ; est-ce que l'octet chargé est 0?
25         BNE      |L0.12|        ; sauter au début de la boucle, si ce n'est pas 0
26 |L0.38|
27 ; renvoyer "s"
28         MOVS     r0,r4
29         POP      {r4,pc}
30         ENDP
```

3.18.8 MIPS

Listing 3.72 : GCC 4.4.5 avec optimisation (IDA)

```
str_trim:
; IDA n'a pas connaissance des noms des variables locales, nous les entrons
    manuellement
saved_GP          = -0x10
saved_S0           = -8
saved_RA           = -4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+saved_RA($sp)
                sw      $s0, 0x20+saved_S0($sp)
                sw      $gp, 0x20+saved_GP($sp)
; appeler strlen(). l'adresse de la chaîne en entrée est toujours dans $a0,
; strlen() la prendra d'ici:
                lw      $t9, (strlen & 0xFFFF)($gp)
                or      $at, $zero ; slot de délai de chargement, NOP
                jalr    $t9
; l'adresse de la chaîne en entrée est toujours dans $a0, mettons la dans
    $s0:
                move    $s0, $a0   ; slot de délai de branchement
; le résultat de strlen() (i.e., la longueur de la chaîne) est maintenant
    dans $v0
; sauter à la sortie si $v0==0 (i.e., la longueur de la chaîne est 0):
                beqz    $v0, exit
                or      $at, $zero ; slot de délai de branchement, NOP
                addiu   $a1, $v0, -1
; $a1 = $v0-1 = str_len-1
                addu    $a1, $s0, $a1
; $a1 = adresse de la chaîne en entrée + $a1 = s+strlen-1
; charger l'octet à l'adresse $a1:
                lb      $a0, 0($a1)
                or      $at, $zero ; slot de délai de chargement, NOP
; est-ce que l'octet est zéro? sauter à la sortie si oui:
                beqz    $a0, exit
                or      $at, $zero ; slot de délai de branchement, NOP
                addiu   $v1, $v0, -2
; $v1 = str_len-2
                addu    $v1, $s0, $v1
; $v1 = $s0+$v1 = s+str_len-2
                li      $a2, 0xD
; sauter le corps de boucle:
                b       loc_6C
                li      $a3, 0xA   ; slot de délai de branchement
loc_5C:
; charger l'octet suivant de la mémoire dans $a0:
                lb      $a0, 0($v1)
                move    $a1, $v1
; $a1=s+str_len-2
; sauter à la sortie si l'octet chargé est zéro:
                beqz    $a0, exit
```

```
; décrémenter str_len:
                addiu   $v1, -1    ; slot de délai de branchement
loc_6C:
; à ce moment, $a0=octet chargé, $a2=0xD (symbole CR) et $a3=0xA (symbole LF)
; l'octet chargé est CR? sauter alors en loc_7C:
                beq     $a0, $a2, loc_7C
                addiu   $v0, -1    ; slot de délai de branchement
; l'octet chargé est LF? sauter à la sortie si ce n'est pas LF:
                bne     $a0, $a3, exit
                or      $at, $zero ; slot de délai de branchement, NOP
loc_7C:
; l'octet chargé est CR à ce moment
; sauter en loc_5c (début du corps de la boucle) si str_len (dans $v0) n'est
    pas zéro:
                bnez    $v0, loc_5C
; simultanément, stocker zéro à cet endroit en mémoire:
                sb      $zero, 0($a1) ; slot de délai de branchement
; le label "exit" à été renseigné manuellemnt:
exit:
                lw      $ra, 0x20+saved_RA($sp)
                move    $v0, $s0
                lw      $s0, 0x20+saved_S0($sp)
                jr      $ra
                addiu   $sp, 0x20     ; slot de délai de branchement
```

Les registres préfixés avec S- sont aussi appelés «saved temporaries» (sauvé temporairement), donc la valeur de $S0 est sauvée dans la pile locale et restaurée à la fin.

3.19 Fonction toupper()

Une autre fonction courante transforme un symbole de minuscule en majuscule, si besoin:

```
char toupper (char c)
{
        if(c>='a' && c<='z')
                return c-'a'+'A';
        else
                return c;
}
```

L'expression `'a'+'A'` est laissée dans le code source pour améliorer la lisibilité, elle sera optimisée par le compilateur, bien sûr. [21].

Le code ASCII de «a» est 97 (ou 0x61), et celui de «A», 65 (ou 0x41).

La différence (ou distance) entre les deux dans la table ASCII est 32 (ou 0x20).

[21]Toutefois, pour être méticuleux, il y a toujours des compilateurs qui ne peuvent pas optimiser de telles expressions et les laissent telles quelles dans le code.

Pour une meilleure compréhension, le lecteur peut regarder la table ASCII 7-bit standard:

```
Characters in the coded character set ascii.
---------------------------------------------------------------------------
      0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
  0x C-@ C-a C-b C-c C-d C-e C-f C-g C-h TAB C-j C-k C-l RET C-n C-o
  1x C-p C-q C-r C-s C-t C-u C-v C-w C-x C-y C-z ESC C-\ C-] C-^ C-_
  2x      !   "   #   $   %   &   '   (   )   *   +   ,   -   .   /
  3x 0    1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?
  4x @    A   B   C   D   E   F   G   H   I   J   K   L   M   N   O
  5x P    Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^
  6x `    a   b   c   d   e   f   g   h   i   j   k   l   m   n   o
  7x p    q   r   s   t   u   v   w   x   y   z   {   |   }   ~   DEL
```

Fig. 3.3: table ASCII 7-bit dans Emacs

3.19.1 x64

Deux opérations de comparaison

MSVC sans optimisation est direct: le code vérifie si le symbole en entrée est dans l'intervalle [97..122] (ou dans l'intervalle ['a'..'z']) et soustrait 32 si c'est le cas.

Il y a quelques artefacts du compilateur:

Listing 3.73 : MSVC 2013 (x64) sans optimisation

```
1  c$ = 8
2  toupper PROC
3          mov     BYTE PTR [rsp+8], cl
4          movsx   eax, BYTE PTR c$[rsp]
5          cmp     eax, 97
6          jl      SHORT $LN2@toupper
7          movsx   eax, BYTE PTR c$[rsp]
8          cmp     eax, 122
9          jg      SHORT $LN2@toupper
10         movsx   eax, BYTE PTR c$[rsp]
11         sub     eax, 32
12         jmp     SHORT $LN3@toupper
13         jmp     SHORT $LN1@toupper      ; artefact du compilateur
14 $LN2@toupper:
15         movzx   eax, BYTE PTR c$[rsp]   ; casting inutile
16 $LN1@toupper:
17 $LN3@toupper:                           ; artefact du compilateur
18         ret     0
19 toupper ENDP
```

Il est important de remarquer que l'octet en entrée est chargé dans un slot 64-bit de la pile locale à la ligne 3.

Tous les bits restants ([8..e3]) ne sont pas touchés, i.e., contiennent du bruit indéterminé (vous le verrez dans le débogueur).

Toutes les instructions opèrent seulement au niveau de l'octet, donc c'est bon.

La dernière instruction MOVZX à la ligne 15 prend un octet de la pile locale et l'étend avec des zéro à un type de donnée *int* 32-bit.

GCC sans optimisation fait essentiellement la même chose:

Listing 3.74 : GCC 4.9 (x64) sans optimisation

```
toupper:
        push    rbp
        mov     rbp, rsp
        mov     eax, edi
        mov     BYTE PTR [rbp-4], al
        cmp     BYTE PTR [rbp-4], 96
        jle     .L2
        cmp     BYTE PTR [rbp-4], 122
        jg      .L2
        movzx   eax, BYTE PTR [rbp-4]
        sub     eax, 32
        jmp     .L3
.L2:
        movzx   eax, BYTE PTR [rbp-4]
.L3:
        pop     rbp
        ret
```

Une opération de comparaison

MSVC avec optimisation fait un meilleur travail, il ne génère qu'une seule opération de comparaison:

Listing 3.75 : MSVC 2013 (x64) avec optimisation

```
toupper PROC
        lea     eax, DWORD PTR [rcx-97]
        cmp     al, 25
        ja      SHORT $LN2@toupper
        movsx   eax, cl
        sub     eax, 32
        ret     0
$LN2@toupper:
        movzx   eax, cl
        ret     0
toupper ENDP
```

Il a déjà été expliqué comment remplacer les deux opérations de comparaison par une seule: 3.13.2 on page 647.

Nous allons maintenant récrire ceci en C/C++ :

```
int tmp=c-97;

if (tmp>25)
        return c;
else
        return c-32;
```

La variable *tmp* doit être signée.

Cela fait deux opérations de soustraction en cas de transformation plus une comparaison.

Par contraste, l'algorithme original utilise deux opérations de comparaison plus une soustraction.

GCC avec optimisation est encore meilleur, il supprime le saut (ce qui est bien: 2.4.1 on page 589) en utilisant l'instruction CMOVcc:

Listing 3.76 : GCC 4.9 (x64) avec optimisation

```
1 toupper:
2         lea     edx, [rdi-97] ; 0x61
3         lea     eax, [rdi-32] ; 0x20
4         cmp     dl, 25
5         cmova   eax, edi
6         ret
```

À la ligne 3 le code prépare la valeur soustraite en avance, comme si la conversion avait toujours lieu.

À la ligne 5 la valeur soustraite dans EAX est remplacée par la valeur en entrée non modifiée si la conversion n'est pas nécessaire. Et ensuite cette valeur (bien sûr incorrecte) est abandonnée.

La soustraction en avance est le prix que le compilateur paye pour l'absence de saut conditionnel.

3.19.2 ARM

Keil avec optimisation pour le mode ARM génère aussi une seule comparaison:

Listing 3.77 : avec optimisation Keil 6/2013 (Mode ARM)

```
toupper PROC
        SUB      r1,r0,#0x61
        CMP      r1,#0x19
        SUBLS    r0,r0,#0x20
        ANDLS    r0,r0,#0xff
        BX       lr
        ENDP
```

Les instructions SUBLS et ANDLS ne sont exécutées que si la valeur dans R1 est inférieure à 0x19 (ou égale).

Keil avec optimisation pour le mode Thumb génère lui aussi une seule opération de comparaison:

Listing 3.78 : avec optimisation Keil 6/2013 (Mode Thumb)

```
toupper PROC
        MOVS     r1,r0
        SUBS     r1,r1,#0x61
```

```
        CMP      r1,#0x19
        BHI      |L0.14|
        SUBS     r0,r0,#0x20
        LSLS     r0,r0,#24
        LSRS     r0,r0,#24
|L0.14|
        BX       lr
        ENDP
```

Les deux dernières instructions LSLS et LSRS fonctionnent comme `AND reg, 0xFF` : elles sont équivalentes à l'expression C/C++ $(i << 24) >> 24$.

Il semble que Keil pour le mode Thumb déduit que ces deux instructions de 2-octets sont plus courtes que le code qui charge la constante 0xFF dans un registre plus une instruction AND.

GCC pour ARM64

Listing 3.79 : GCC 4.9 (ARM64) sans optimisation

```
toupper:
        sub     sp, sp, #16
        strb    w0, [sp,15]
        ldrb    w0, [sp,15]
        cmp     w0, 96
        bls     .L2
        ldrb    w0, [sp,15]
        cmp     w0, 122
        bhi     .L2
        ldrb    w0, [sp,15]
        sub     w0, w0, #32
        uxtb    w0, w0
        b       .L3
.L2:
        ldrb    w0, [sp,15]
.L3:
        add     sp, sp, 16
        ret
```

Listing 3.80 : GCC 4.9 (ARM64) avec optimisation

```
toupper:
        uxtb    w0, w0
        sub     w1, w0, #97
        uxtb    w1, w1
        cmp     w1, 25
        bhi     .L2
        sub     w0, w0, #32
        uxtb    w0, w0
.L2:
        ret
```

3.19.3 Utilisation d'opérations sur les bits

Étant donné le fait que le bit d'indice 5 (en partant depuis 0) est toujours présent après le test, soustraire revient juste à effacer ce seul bit, mais la même chose peut être effectuée avec un AND.

Encore plus simple, en XOR-ant:

```
char toupper (char c)
{
        if(c>='a' && c<='z')
                return c^0x20;
        else
                return c;
}
```

Le code est proche de ce GCC avec optimisation a produit pour l'exemple précédent (3.76 on page 689) :

Listing 3.81 : GCC 5.4 (x86) avec optimisation

```
toupper:
        mov     edx, DWORD PTR [esp+4]
        lea     ecx, [edx-97]
        mov     eax, edx
        xor     eax, 32
        cmp     cl, 25
        cmova   eax, edx
        ret
```

…mais XOR est utilisé au lieu de SUB.

Changer le bit d'indice 5 est juste déplacer un *curseur* dans la table ASCII en haut ou en bas de deux lignes.

Certains disent que les lettres minuscules/majuscules ont été placées de cette façon dans la table ASCII intentionnellement, car:

> Very old keyboards used to do Shift just by toggling the 32 or 16 bit, depending on the key; this is why the relationship between small and capital letters in ASCII is so regular, and the relationship between numbers and symbols, and some pairs of symbols, is sort of regular if you squint at it.

(Eric S. Raymond, `http://www.catb.org/esr/faqs/things-every-hacker-once-knew/`)

Donc, nous pouvons écrire ce morceau de code, qui change juste la casse des lettres:

```
#include <stdio.h>

char flip (char c)
{
        if((c>='a' && c<='z') || (c>='A' && c<='Z'))
```

```
                return c^0x20;
        else
                return c;
}

int main()
{
        // affichera "hELLO, WORLD!"
        for (char *s="Hello, world!"; *s; s++)
                printf ("%c", flip(*s));
};
```

3.19.4 Summary

Toutes ces optimisations de compilateurs sont aujourd'hui courantes et un rétro-ingénieur pratiquant voit souvent ce genre de patterns de code.

3.20 Obfuscation

L'obfuscation est une tentative de cacher le code (ou sa signification) aux rétro-ingénieurs.

3.20.1 Chaînes de texte

Comme nous l'avons vu dans (5.4 on page 914), les chaînes de texte peuvent être vraiment utiles.

Les programmeurs qui sont conscients de ceci essayent de les cacher, rendant impossible de trouver la chaîne dans IDA ou tout autre éditeur hexadécimal.

Voici la méthode la plus simple.

La chaîne peut être construite comme ceci:

```
mov     byte ptr [ebx], 'h'
mov     byte ptr [ebx+1], 'e'
mov     byte ptr [ebx+2], 'l'
mov     byte ptr [ebx+3], 'l'
mov     byte ptr [ebx+4], 'o'
mov     byte ptr [ebx+5], ' '
mov     byte ptr [ebx+6], 'w'
mov     byte ptr [ebx+7], 'o'
mov     byte ptr [ebx+8], 'r'
mov     byte ptr [ebx+9], 'l'
mov     byte ptr [ebx+10], 'd'
```

La chaîne peut aussi être comparée avec une autre comme ceci:

```
mov     ebx, offset username
cmp     byte ptr [ebx], 'j'
jnz     fail
```

```
cmp     byte ptr [ebx+1], 'o'
jnz     fail
cmp     byte ptr [ebx+2], 'h'
jnz     fail
cmp     byte ptr [ebx+3], 'n'
jnz     fail
jz      it_is_john
```

Dans les deux cas, il est impossible de trouver ces chaînes directement dans un éditeur hexadécimal.

À propos, ceci est un moyen de travailler avec des chaînes lorsqu'il est impossible d'allouer de l'espace pour elles dans le segment de données, par exemple dans un PIC[22] ou un shellcode.

Une autre méthode est d'utiliser `sprintf()` pour la construction:

```
sprintf(buf, "%s%c%s%c%s", "hel",'l',"o w",'o',"rld");
```

Le code semble bizarre, mais peut être utile comme simple mesure anti-reversing.

Les chaînes de texte peuvent aussi être présentes dans une forme chiffrée, donc chaque utilisation d'une chaîne est précédée par une routine de déchiffrement. Par exemple: 8.8.2 on page 1091.

3.20.2 Code exécutable

Insertion de code inutile

L'obfuscation de code exécutable implique l'insertion aléatoire de code inutile dans le code réel, qui s'exécute mais ne fait rien d'utile.

Un simple exemple:

Listing 3.82 : code original

```
add     eax, ebx
mul     ecx
```

Listing 3.83 : code obfusqué

```
xor     esi, 011223344h ; inutile
add     esi, eax        ; inutile
add     eax, ebx
mov     edx, eax        ; inutile
shl     edx, 4          ; inutile
mul     ecx
xor     esi, ecx        ; inutile
```

Ici, le code inutile utilise des registres qui ne sont pas utilisés dans le code réel (ESI et EDX). Toutefois, les résultats intermédiaires produit par le code réel peuvent être utilisés par les instructions inutiles pour brouiller les pistes—pourquoi pas?

[22] Position Independent Code

Remplacer des instructions avec des équivalents plus gros

- `MOV op1, op2` peut être remplacé par la paire `PUSH op2 / POP op1`.
- `JMP label` peut être remplacé par la paire `PUSH label / RET`. IDA ne montrera pas la référence au label.
- `CALL label` peut être remplacé par le triplet d'instructions suivant: `PUSH label_after_CALL_instruction / PUSH label / RET`.
- `PUSH op` peut aussi être remplacé par la paire d'instructions suivante: `SUB ESP, 4 (or 8) / MOV [ESP], op`.

Code toujours exécuté/jamais exécuté

Si le développeur est certain que ESI contient toujours 0 à ce point:

```
mov     esi, 1
...     ; du code qui ne change pas ESI
dec     esi
...     ; du code qui ne change pas ESI
cmp     esi, 0
jz      real_code
; fake code
real_code:
```

Le rétro-ingénieur a parfois besoin de temps pour le comprendre.

Ceci est aussi appelé un *prédicat opaque*.

Un autre exemple (et de nouveau, le développeur est certain que ESI vaut toujours zéro) :

```
; ESI=0
add     eax, ebx        ; code réel
mul     ecx             ; code réel
add     eax, esi        ; prédicat opaque.
              ; XOR, AND ou SHL, etc, peuvent être ici au lieu de ADD.
```

Mettre beaucoup de bazar

```
instruction 1
instruction 2
instruction 3
```

Peut être remplacé par:

```
begin:          jmp     ins1_label

ins2_label:     instruction 2
                jmp     ins3_label

ins3_label:     instruction 3
                jmp     exit:
```

```
ins1_label:     instruction 1
                jmp     ins2_label
exit:
```

Utilisation de pointeurs indirects

```
dummy_data1     db      100h dup (0)
message1        db      'hello world',0

dummy_data2     db      200h dup (0)
message2        db      'another message',0

func            proc
                ...
                mov     eax, offset dummy_data1 ; PE or ELF reloc here
                add     eax, 100h
                push    eax
                call    dump_string
                ...
                mov     eax, offset dummy_data2 ; PE or ELF reloc here
                add     eax, 200h
                push    eax
                call    dump_string
                ...
func            endp
```

IDA montrera seulement les références à dummy_data1 et dummy_data2, mais pas aux chaînes de textes.

Les variables globales et même les fonctions peuvent être accédées comme ça.

Maintenant, quelque chose de légèrement plus avancé.

Franchement, je ne connais pas con nom exact, mais je vais l'appeler *pointeur décalés*. Cette technique est assez commune, au moins dans les systèmes de protection contre la copie.

En bref: lorsque vous écrivez une valeur dans la mémoire globale, vous utilisez une adresse, mais lorsque vous lisez, vous utilisez la somme d'une (autre) adresse, ou peut-être une différence. Le but est de cacher l'adresse réelle au rétro-ingénieur qui débogue le code ou l'explore dans IDA (ou un autre désassembleur).

Ceci peut être pénible.

```
#include <stdio.h>

// 64KiB, but it's OK
unsigned char secret_array[0x10000];

void check_lic_key()
{
        // pretend licence check has been failed
```

```
        secret_array[0x6123]=1; // 1 mean failed

        printf ("check failed\n"); //  exit(0); // / a cracker may patch here

        // or put there another value if check is succeeded
        secret_array[0x6123]=0;
};

unsigned char get_byte_at_0x6000(unsigned char *a)
{
        return *(a+0x6000);
};

void check_again()
{
        if (get_byte_at_0x6000(secret_array+0x123)==1)
        {
                // do something mean (add watermark maybe) or report error:
                printf ("check failed\n");
        }
        else
        {
                // proceed further
        };
};

int main()
{
        // at start:
        check_lic_key();

        // do something

        // ... and while in some very critical part:
        check_again();
};
```

Compiler avec MSVC 2015 sans optimisation:

```
_check_lic_key  proc near
                push    ebp
                mov     ebp, esp
                mov     eax, 1
                imul    ecx, eax, 6123h
                mov     _secret_array[ecx], 1
                pop     ebp
                retn
_check_lic_key  endp


_get_byte_at_0x6000 proc near

a               = dword ptr  8

```

```
                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+a]
                mov     al, [eax+6000h]
                pop     ebp
                retn
_get_byte_at_0x6000 endp

_check_again    proc near
                push    ebp
                mov     ebp, esp
                push    offset point_passed_to_get_byte_at_0x6000
                call    j__get_byte_at_0x6000
                add     esp, 4
                movzx   eax, al
                cmp     eax, 1
                jnz     short loc_406735
                push    offset _Format  ; "check failed\n"
                call    j__printf
                add     esp, 4

loc_406735:
                pop     ebp
                retn
_check_again    endp

.data:0045F5C0 ; char secret_array[65536]
.data:0045F5C0 _secret_array   db 123h dup(?)
.data:0045F6E3 ; char point_passed_to_get_byte_at_0x6000[65245]
.data:0045F6E3 point_passed_to_get_byte_at_0x6000 db 0FEDDh dup(?)
```

Vous voyez, IDA obtient seulement deux adresses: `secret_array[]` (début du tableau) et `point_passed_to_get_byte_at_0x6000`.

Comment s'y prendre: vous pouvez utiliser les point d'arrêt matériel sur les opérations d'accès en mémoire, tracer possède l'option BPMx) ou des moteurs d'exécution symbolique ou peut-être écrire un module pour IDA …

C'est sûr, un tableau peut être utiliser pour des nombreuses valeurs, non limité aux booléennes…

N.B.: MSVC 2015 avec optimisation est assez malin pour optimiser la fonction `get_byte_at_0x6000()`.

3.20.3 Machine virtuelle / pseudo-code

Un programmeur peut construire son propre LP ou ISA et son interpréteur.

(Comme le Visual Basic pre-5.0, .NET ou les machines Java). Le rétro-ingénieur aura besoin de passer du temps pour comprendre la signification et les détails de toutes les instructions de l'ISA.

Il/elle devra aussi une sorte de désassembleur/décompilateur.

3.20.4 Autres choses à mentionner

Ma propre (faible pour le moment) tentative de patch du compilateur Tiny C pour produire du code obfusqué: http://blog.yurichev.com/node/58.

Utiliser l'instruction MOV pour des choses vraiment compliquées: [Stephen Dolan, *mov is Turing-complete*, (2013)] [23].

3.20.5 Exercice

- http://challenges.re/29

3.21 C++

3.21.1 Classes

Un exemple simple

En interne, la représentation des classes C++ est presque la même que les structures.

Essayons un exemple avec deux variables, deux constructeurs et une méthode:

```
#include <stdio.h>

class c
{
private:
    int v1;
    int v2;
public:
    c() // ctor par défaut
    {
        v1=667;
        v2=999;
    };

    c(int a, int b) // ctor
    {
        v1=a;
        v2=b;
    };

    void dump()
    {
        printf ("%d; %d\n", v1, v2);
    };
};

int main()
{
```

[23] Aussi disponible en http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf

```
    class c c1;
    class c c2(5,6);

    c1.dump();
    c2.dump();

    return 0;
};
```

MSVC: x86

Voici à quoi ressemble la fonction `main()`, traduite en langage d'assemblage:

Listing 3.84 : MSVC

```
_c2$ = -16 ; size = 8
_c1$ = -8  ; size = 8
_main PROC
    push ebp
    mov  ebp, esp
    sub  esp, 16
    lea  ecx, DWORD PTR _c1$[ebp]
    call ??0c@@QAE@XZ ; c::c
    push 6
    push 5
    lea  ecx, DWORD PTR _c2$[ebp]
    call ??0c@@QAE@HH@Z ; c::c
    lea  ecx, DWORD PTR _c1$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    lea  ecx, DWORD PTR _c2$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    xor  eax, eax
    mov  esp, ebp
    pop  ebp
    ret  0
_main ENDP
```

Voici ce qui se passe. Pour chaque objet (instance de la classe c) 8 octets sont alloués, exactement la taille requise pour stocker les deux variables.

Pour *c1* un constructeur par défaut sans argument `??0c@@QAE@XZ` est appelé. Pour *c2* un autre constructeur `??0c@@QAE@HH@Z` est appelé et deux nombres sont passés comme arguments.

Un pointeur sur l'objet (*this* en terminologie C++) est passé dans le registre ECX. Ceci est appelé thiscall (3.21.1)—la méthode pour passer un pointeur à l'objet.

MSVC le fait en utilisant le regsitre ECX. Inutile de le dire, ce n'est pas une méthode standardisée, d'autres compilateurs peuvent le faire différemment, e.g., par le premier argument de la fonction (comme GCC).

Pourquoi est-ce que ces fonctions ont un nom aussi étrange? C'est le name mangling.

Une classe C++ peut contenir plusieurs méthodes partageant le même nom mais ayant des arguments différents—c'est le polymorphisme. Et bien sûr, différentes classes peuvent avoir leurs propres méthodes avec le même nom.

Le *Name mangling* nous permet d'encoder le nom de la classe + le nom de la méthode + tous les types des arguments de la méthode dans une chaîne ASCII, qui est ensuite utilisée comme le nom interne de la fonction. C'est ainsi car ni le linker, ni le chargeur de DLL de l'OS (les mangled names peuvent aussi se trouver parmi les exports de DLL) n'ont conscience de C++ ou de l'POO[24].

La fonction dump() est appelée deux fois.

Maintenant, regardons le code du constructeur:

Listing 3.85 : MSVC

```
_this$ = -4        ; size = 4
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  DWORD PTR [eax], 667
    mov  ecx, DWORD PTR _this$[ebp]
    mov  DWORD PTR [ecx+4], 999
    mov  eax, DWORD PTR _this$[ebp]
    mov  esp, ebp
    pop  ebp
    ret  0
??0c@@QAE@XZ ENDP ; c::c

_this$ = -4 ; size = 4
_a$ = 8     ; size = 4
_b$ = 12    ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR _a$[ebp]
    mov  DWORD PTR [eax], ecx
    mov  edx, DWORD PTR _this$[ebp]
    mov  eax, DWORD PTR _b$[ebp]
    mov  DWORD PTR [edx+4], eax
    mov  eax, DWORD PTR _this$[ebp]
    mov  esp, ebp
    pop  ebp
    ret  8
??0c@@QAE@HH@Z ENDP ; c::c
```

[24] Programmation orientée objet

Les constructeurs sont juste des fonctions, ils utilisent un pointeur sur la structure dans ECX, en copiant le pointeur dans leur propre variable locale, toutefois, ce n'est pas nécessaire.

D'après le standard (C++11 12.1) nous savons que les constructeurs n'ont pas l'obligation de renvoyer une valeur.

En fait, en interne, les constructeurs renvoient un pointeur sur l'objet nouvellement créé, i.e., *this*.

Maintenant, la méthode `dump()` :

Listing 3.86 : MSVC

```
_this$ = -4          ; size = 4
?dump@c@@QAEXXZ PROC ; c::dump, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR [eax+4]
    push ecx
    mov  edx, DWORD PTR _this$[ebp]
    mov  eax, DWORD PTR [edx]
    push eax
    push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
    call _printf
    add  esp, 12
    mov  esp, ebp
    pop  ebp
    ret  0
?dump@c@@QAEXXZ ENDP ; c::dump
```

Assez simple: `dump()` prend un pointeur sur la structure qui contient les deux *int* dans ECX, prend les deux valeurs et les passe à `printf()`.

Le code est bien plus court s'il est compilé avec les optimisations (/Ox) :

Listing 3.87 : MSVC

```
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
    mov  eax, ecx
    mov  DWORD PTR [eax], 667
    mov  DWORD PTR [eax+4], 999
    ret  0
??0c@@QAE@XZ ENDP ; c::c

_a$ = 8  ; size = 4
_b$ = 12 ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
    mov  edx, DWORD PTR _b$[esp-4]
    mov  eax, ecx
```

```
    mov  ecx, DWORD PTR _a$[esp-4]
    mov  DWORD PTR [eax], ecx
    mov  DWORD PTR [eax+4], edx
    ret  8
??0c@@QAE@HH@Z ENDP ; c::c

?dump@c@@QAEXXZ PROC ; c::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push eax
    push ecx
    push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
    call _printf
    add  esp, 12
    ret  0
?dump@c@@QAEXXZ ENDP ; c::dump
```

C'est tout. L'autre chose que nous devons noter est que le pointeur de pile n'a pas été corrigé avec `add esp, X` après l'appel du constructeur. En même temps, le constructeur a `ret 8` au lieu de RET à la fin.

C'est parce que la convention d'appel thiscall (3.21.1 on page 699) est utilisée ici, qui, comme avec la méthode stdcall (6.1.2 on page 953), offre à l'appelée la possibilité de corriger la pile au lieu de l'appelante. L'instruction `ret x` ajoute X à la valeur de ESP, puis passe le contrôle à la fonction appelante.

Regardez la section parlant des conventions d'appel (6.1 on page 953).

Il faut également noter que le compilateur décide quand appeler le constructeur et le destructeur—mais nous le savons déjà des bases du langage C++.

MSVC: x86-64

Comme nous le savons déjà, les 4 premiers arguments de fonction en x86-64 sont passés dans les registres RCX, RDX, R8 et R9, tous les autres—par la pile.

Néanmoins, le pointeur *this* sur l'objet est passé dans RCX, le premier argument de la méthode dans RDX, etc. Nous pouvons le voir dans les entrailles de la méthode `c(int a, int b)` :

Listing 3.88 : MSVC 2012 x64 avec optimisation

```
; void dump()

?dump@c@@QEAAXXZ PROC ; c::dump
    mov    r8d, DWORD PTR [rcx+4]
    mov    edx, DWORD PTR [rcx]
    lea    rcx, OFFSET FLAT:??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@ ;
    '%d; %d'
    jmp    printf
?dump@c@@QEAAXXZ ENDP ; c::dump
```

```
; c(int a, int b)

??0c@@QEAA@HH@Z PROC ; c::c
    mov    DWORD PTR [rcx], edx   ; 1er argument: a
    mov    DWORD PTR [rcx+4], r8d ; 2nd argument: b
    mov    rax, rcx
    ret    0
??0c@@QEAA@HH@Z ENDP ; c::c

; default ctor

??0c@@QEAA@XZ PROC ; c::c
    mov    DWORD PTR [rcx], 667
    mov    DWORD PTR [rcx+4], 999
    mov    rax, rcx
    ret    0
??0c@@QEAA@XZ ENDP ; c::c
```

Le type de donnée *int* est toujours 32-bit en x64 [25], c'est donc pourquoi les parties 32-bit des registres sont utilisées ici.

Nous voyons également JMP `printf` au lieu de RET dans la méthode `dump()`, *astuce* que nous avons déjà vu plus tôt: 1.21.1 on page 205.

GCC: x86

C'est presque la même chose avec GCC 4.4.1, avec quelques exceptions.

Listing 3.89 : GCC 4.4.1

```
     public main
main proc near

var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
var_18 = dword ptr -18h
var_10 = dword ptr -10h
var_8  = dword ptr -8

    push ebp
    mov  ebp, esp
    and  esp, 0FFFFFFF0h
    sub  esp, 20h
    lea  eax, [esp+20h+var_8]
    mov  [esp+20h+var_20], eax
    call _ZN1cC1Ev
    mov  [esp+20h+var_18], 6
    mov  [esp+20h+var_1C], 5
    lea  eax, [esp+20h+var_10]
    mov  [esp+20h+var_20], eax
    call _ZN1cC1Eii
```

[25] Apparemment, pour faciliter le portage de code 32-bit C/C++ en x64

```
    lea  eax, [esp+20h+var_8]
    mov  [esp+20h+var_20], eax
    call _ZN1c4dumpEv
    lea  eax, [esp+20h+var_10]
    mov  [esp+20h+var_20], eax
    call _ZN1c4dumpEv
    mov  eax, 0
    leave
    retn
main endp
```

Ici, nous voyons un autre style de *name mangling*, spécifique à GNU [26]. Il peut aussi être noté que le pointeur sur l'objet est passé comme premier argument de la fonction—invisible au programmeur, bien sûr.

Premier constructeur:

```
                public _ZN1cC1Ev ; weak
_ZN1cC1Ev       proc near               ; CODE XREF: main+10

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                mov     dword ptr [eax], 667
                mov     eax, [ebp+arg_0]
                mov     dword ptr [eax+4], 999
                pop     ebp
                retn
_ZN1cC1Ev       endp
```

Il écrit juste les deux nombres en utilisant le pointeur passé comme premier (et seul) argument.

Second constructeur:

```
                public _ZN1cC1Eii
_ZN1cC1Eii      proc near

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                mov     edx, [ebp+arg_4]
                mov     [eax], edx
                mov     eax, [ebp+arg_0]
```

[26] Il y a un bon document à propos des différentes conventions de name mangling dans différent compilateurs:
[Agner Fog, *Calling conventions* (2015)].

```
                mov     edx, [ebp+arg_8]
                mov     [eax+4], edx
                pop     ebp
                retn
_ZN1cC1Eii      endp
```

Ceci est une fonction, l’analogue d’une qui pourrait ressembler à ceci:

```
void ZN1cC1Eii (int *obj, int a, int b)
{
  *obj=a;
  *(obj+1)=b;
};
```

…et cela est entièrement prévisible.

Maintenant, la fonction `dump()` :

```
                public _ZN1c4dumpEv
_ZN1c4dumpEv    proc near

var_18          = dword ptr -18h
var_14          = dword ptr -14h
var_10          = dword ptr -10h
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 18h
                mov     eax, [ebp+arg_0]
                mov     edx, [eax+4]
                mov     eax, [ebp+arg_0]
                mov     eax, [eax]
                mov     [esp+18h+var_10], edx
                mov     [esp+18h+var_14], eax
                mov     [esp+18h+var_18], offset aDD ; "%d; %d\n"
                call    _printf
                leave
                retn
_ZN1c4dumpEv    endp
```

La *représentation interne* de cette fonction a un seul argument, utilisé comme pointeur sur l’objet (*this*).

Cette fonction pourrait être récrite en C comme ceci:

```
void ZN1c4dumpEv (int *obj)
{
  printf ("%d; %d\n", *obj, *(obj+1));
};
```

Ainsi, si nous basons notre jugement sur ces simples exemples, la différence entre MSVC et GCC est le style d’encodage des noms de fonctions (*name mangling*) et la

méthode pour passer un pointeur sur l'objet (via le registre ECX ou via le premier argument).

GCC: x86-64

Les 6 premiers arguments, comme nous le savons déjà, sont passés par les registres RDI, RSI, RDX, RCX, R8 et R9 ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [27]), et le pointeur sur *this* via le premier (RDI) et c'est ce que l'on voit ici. Le type de donnée *int* est aussi 32-bit ici.

L'*astuce* du JMP au lieu de RET est aussi utilisée ici.

Listing 3.90 : GCC 4.4.6 x64

```
; ctor par défault

_ZN1cC2Ev:
    mov  DWORD PTR [rdi], 667
    mov  DWORD PTR [rdi+4], 999
    ret

; c(int a, int b)

_ZN1cC2Eii:
    mov  DWORD PTR [rdi], esi
    mov  DWORD PTR [rdi+4], edx
    ret

; dump()

_ZN1c4dumpEv:
    mov  edx, DWORD PTR [rdi+4]
    mov  esi, DWORD PTR [rdi]
    xor  eax, eax
    mov  edi, OFFSET FLAT:.LC0 ; "%d; %d\n"
    jmp  printf
```

Héritage de classe

Les classes héritées sont similaires aux simples structures dont nous avons déjà discuté, mais étendues aux classes héritables.

Prenons ce simple exemple:

```
#include <stdio.h>

class object
{
    public:
```

[27] Aussi disponible en https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

```
        int color;
        object() { };
        object (int color) { this->color=color; };
        void print_color() { printf ("color=%d\n", color); };
};

class box : public object
{
    private:
        int width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is a box. color=%d, width=%d, height=%d, depth=%d↙
    ↘ \n", color, width, height, depth);
        };
};

class sphere : public object
{
private:
    int radius;
public:
    sphere(int color, int radius)
    {
        this->color=color;
        this->radius=radius;
    };
    void dump()
    {
        printf ("this is sphere. color=%d, radius=%d\n", color, radius);
    };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    b.print_color();
    s.print_color();

    b.dump();
    s.dump();

    return 0;
```

```
};
```

Investiguons le code généré de la fonction/méthode `dump()` et aussi `object::print_color()`, et regardons la disposition de la mémoire pour les structures-objets (pour du code 32-bit).

Donc, voici les méthodes `dump()` pour quelques classes, générées par MSVC 2008 avec les options `/Ox` et `/Ob0`[28].

Listing 3.91 : MSVC 2008 avec optimisation /Ob0

```
??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@ DB 'color=%d', 0aH, 00H ; `string'
?print_color@object@@QAEXXZ PROC ; object::print_color, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx]
    push eax

; 'color=%d', 0aH, 00H
    push OFFSET ??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@
    call _printf
    add  esp, 8
    ret  0
?print_color@object@@QAEXXZ ENDP ; object::print_color
```

Listing 3.92 : MSVC 2008 avec optimisation /Ob0

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+12]
    mov  edx, DWORD PTR [ecx+8]
    push eax
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx

;
    'this is a box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H ; `string'
    push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0?5width⤦
    ↳ ?$DN?$CFd?0@
    call _printf
    add  esp, 20
    ret  0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Listing 3.93 : MSVC 2008 avec optimisation /Ob0

```
?dump@sphere@@QAEXXZ PROC ; sphere::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+4]
```

[28] L'option `/Ob0` signifie la désactivation de l'expension inline, puisque la mise en ligne de fonctions peut rendre notre expérience plus difficile.

```
    mov  ecx, DWORD PTR [ecx]
    push eax
    push ecx

; 'this is sphere. color=%d, radius=%d', 0aH, 00H
    push OFFSET ??_C@_0CF@EFEDJLDC@this?5is?5sphere?4?5color?$DN?$CFd?0?5↙
    ↘ radius@
    call _printf
    add  esp, 12
    ret  0
?dump@sphere@@QAEXXZ ENDP ; sphere::dump
```

Donc, voici la disposition de la mémoire:

(classe de base *object*)

offset	description
+0x0	int color

(classes héritées)

box :

offset	description
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

sphere :

offset	description
+0x0	int color
+0x4	int radius

Regardons le corps de la fonction `main()` :

Listing 3.94 : MSVC 2008avec optimisation /Ob0

```
PUBLIC _main
_TEXT SEGMENT
_s$ = -24 ; size = 8
_b$ = -16 ; size = 16
_main PROC
    sub  esp, 24
    push 30
    push 20
    push 10
    push 1
    lea  ecx, DWORD PTR _b$[esp+40]
    call ??0box@@QAE@HHHH@Z ; box::box
    push 40
    push 2
    lea  ecx, DWORD PTR _s$[esp+32]
    call ??0sphere@@QAE@HH@Z ; sphere::sphere
    lea  ecx, DWORD PTR _b$[esp+24]
```

```
    call ?print_color@object@@QAEXXZ ; object::print_color
    lea  ecx, DWORD PTR _s$[esp+24]
    call ?print_color@object@@QAEXXZ ; object::print_color
    lea  ecx, DWORD PTR _b$[esp+24]
    call ?dump@box@@QAEXXZ ; box::dump
    lea  ecx, DWORD PTR _s$[esp+24]
    call ?dump@sphere@@QAEXXZ ; sphere::dump
    xor  eax, eax
    add  esp, 24
    ret  0
_main ENDP
```

Les classes héritées doivent toujours ajouter leurs champs après les champs de la classe de base, afin que les méthodes de la classe de base puissent travailler avec ses propres champs.

Lorsque la méthode `object::print_color()` est appelée, un pointeur sur les deux objets *box* et *sphere* est passé en `this`, et il peut travailler facilement avec ces objets puisque le champ *color* dans ces objets est toujours à l'adresse épinglée (à l'offset *+0x0*).

On peut dire que la méthode `object::print_color()` est agnostique en relation avec le type d'objet en entrée tant que les champs sont *épinglés* à la même adresse et cette condition est toujours vraie.

Et si vous créez une classe héritée de la classe *box*, le compilateur ajoutera les nouveaux champs après le champ *depth*, laissant les champs de la classe *box* à l'adresse épinglée.

Ainsi, la méthode `box::dump()` fonctionnera correctement pour accéder aux champs *color*, *width*, *height* et *depth*, qui sont toujours positionnés à l'adresse connue.

Le code généré par GCC est presque le même, avec la seule exception du passage du pointeur `this` (comme il a déjà été expliqué plus haut, il est passé en premier argument au lieu d'utiliser le registre ECX).

Encapsulation

L'encapsulation consiste à cacher les données dans des sections *private* de la classe, e.g. de n'autoriser leurs accès que depuis les méthodes de la classe.

Toutefois, y a-t-il des repères dans le code à propos du fait que certains champs sont privé et d'autres—non?

Non, il n'y a pas de tels repères.

Essayons avec ce simple exemple:

```
#include <stdio.h>

class box
{
    private:
        int color, width, height, depth;
```

```
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is a box. color=%d, width=%d, height=%d, depth=%d↙
    ↳ \n", color, width, height, depth);
        };
};
```

Compilons-le à nouveau dans MSVC 2008 avec les options /Ox et /Ob0 puis regardons le code de la méthode box::dump() :

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+12]
    mov  edx, DWORD PTR [ecx+8]
    push eax
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx
; 'this is a box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H
    push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0?5width↙
    ↳ ?$DN?$CFd?0@
    call _printf
    add  esp, 20
    ret  0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Voici l'agencement mémoire de la classe:

offset	description
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

Tous les champs sont privés et ne peuvent être accédés depuis une autre fonction, mais connaissant cette disposition, pouvons-nous créer le code qui modifie ces champs?

Pour faire ceci, nous ajoutons la fonction hack_oop_encapsulation(), qui ne compilerait pas si elle ressemblait à ceci:

```
void hack_oop_encapsulation(class box * o)
{
    o->width=1; // ce code ne paut pas être compilé:
```

```
                // "error C2248: 'box::width' : cannot access private member
    declared in class 'box'"
};
```

Néanmoins, si nous castons le type *box* sur un *pointeur sur un tableau de int*, que nous modifions le tableau de *int*-s que nous avons, nous pourrions réussir.

```
void hack_oop_encapsulation(class box * o)
{
    unsigned int *ptr_to_object=reinterpret_cast<unsigned int*>(o);
    ptr_to_object[1]=123;
};
```

Le code de cette fonction est très simple—on peut dire que la fonction prend un pointeur sur un tableau de *int*-s en entrée et écrit *123* dans le second *int* :

```
?hack_oop_encapsulation@@YAXPAVbox@@@Z PROC ; hack_oop_encapsulation
    mov eax, DWORD PTR _o$[esp-4]
    mov DWORD PTR [eax+4], 123
    ret 0
?hack_oop_encapsulation@@YAXPAVbox@@@Z ENDP ; hack_oop_encapsulation
```

Regardons comment ça fonctionne:

```
int main()
{
    box b(1, 10, 20, 30);

    b.dump();

    hack_oop_encapsulation(&b);

    b.dump();

    return 0;
};
```

Lançons-le:

```
this is a box. color=1, width=10, height=20, depth=30
this is a box. color=1, width=123, height=20, depth=30
```

Nous voyons que l'encapsulation est juste une protection des champs de la classe lors de l'étape de compilation.

Le compilateur C++ n'autorise pas la génération de code qui modifie directement les champs protégés, néanmoins, il est possible de le faire avec l'aide de *dirty hacks*.

Héritage multiple

L'héritage multiple est la création d'une classe qui hérite des champs et méthodes de deux classes ou plus.

Écrivons à nouveau un exemple simple:

```
#include <stdio.h>

class box
{
    public:
        int width, height, depth;
        box() { };
        box(int width, int height, int depth)
        {
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is a box. width=%d, height=%d, depth=%d\n", width↙
    ↘ , height, depth);
        };
        int get_volume()
        {
            return width * height * depth;
        };
};

class solid_object
{
    public:
        int density;
        solid_object() { };
        solid_object(int density)
        {
            this->density=density;
        };
        int get_density()
        {
            return density;
        };
        void dump()
        {
            printf ("this is a solid_object. density=%d\n", density);
        };
};

class solid_box: box, solid_object
{
    public:
        solid_box (int width, int height, int depth, int density)
        {
            this->width=width;
            this->height=height;
            this->depth=depth;
            this->density=density;
        };
```

```
        void dump()
        {
            printf ("this is a solid_box. width=%d, height=%d, depth=%d, ↙
    ↘ density=%d\n", width, height, depth, density);
        };
        int get_weight() { return get_volume() * get_density(); };
};

int main()
{
    box b(10, 20, 30);
    solid_object so(100);
    solid_box sb(10, 20, 30, 3);

    b.dump();
    so.dump();
    sb.dump();
    printf ("%d\n", sb.get_weight());

    return 0;
};
```

Compilons-le avec MSVC 2008 avec les options `/Ox` et `/Ob0` et regardons le code de `box::dump()`,
`solid_object::dump()` et `solid_box::dump()` :

Listing 3.95 : MSVC 2008 avec optimisation /Ob0

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+8]
    mov  edx, DWORD PTR [ecx+4]
    push eax
    mov  eax, DWORD PTR [ecx]
    push edx
    push eax
; 'this is a box. width=%d, height=%d, depth=%d', 0aH, 00H
    push OFFSET ??_C@_0CM@DIKPHDFI@this?5is?5box?4?5width?$DN?$CFd?0?5↙
    ↘ height?$DN?$CFd@
    call _printf
    add  esp, 16
    ret  0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Listing 3.96 : MSVC 2008 avec optimisation /Ob0

```
?dump@solid_object@@QAEXXZ PROC ; solid_object::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx]
    push eax
; 'this is a solid_object. density=%d', 0aH
    push OFFSET ??_C@_0CC@KICFJINL@this?5is?5solid_object?4?5density?$DN?↙
    ↘ $CFd@
    call _printf
```

```
    add  esp, 8
    ret  0
?dump@solid_object@@QAEXXZ ENDP ; solid_object::dump
```

Listing 3.97 : MSVC 2008 avec optimisation /Ob0

```
?dump@solid_box@@QAEXXZ PROC ; solid_box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+12]
    mov  edx, DWORD PTR [ecx+8]
    push eax
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx
; 'this is a solid_box. width=%d, height=%d, depth=%d, density=%d', 0aH
    push OFFSET ??_C@_0DO@HNCNIHNN@this?5is?5solid_box?4?5width?$DN?$CFd↙
    ↳ ?0?5hei@
    call _printf
    add  esp, 20
    ret  0
?dump@solid_box@@QAEXXZ ENDP ; solid_box::dump
```

Donc, la disposition de la mémoire pour ces trois classes est:

Classe *box* :

offset	description
+0x0	width
+0x4	height
+0x8	depth

Classe *solid_object* :

offset	description
+0x0	density

On peut dire que la disposition de la mémoire de la classe *solid_box* est *unifiée* :

Classe *solid_box* :

offset	description
+0x0	width
+0x4	height
+0x8	depth
+0xC	density

Le code des méthodes `box::get_volume()` et `solid_object::get_density()` est trivial:

Listing 3.98 : MSVC 2008 avec optimisation /Ob0

```
?get_volume@box@@QAEHXZ PROC ; box::get_volume, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+8]
```

```
    imul eax, DWORD PTR [ecx+4]
    imul eax, DWORD PTR [ecx]
    ret  0
?get_volume@box@@QAEHXZ ENDP ; box::get_volume
```

Listing 3.99 : MSVC 2008 avec optimisation /Ob0

```
?get_density@solid_object@@QAEHXZ PROC ; solid_object::get_density, COMDAT
; _this$ = ecx
    mov eax, DWORD PTR [ecx]
    ret 0
?get_density@solid_object@@QAEHXZ ENDP ; solid_object::get_density
```

Mais le code de la méthode `solid_box::get_weight()` est bien plus intéressant:

Listing 3.100 : MSVC 2008 avec optimisation /Ob0

```
?get_weight@solid_box@@QAEHXZ PROC ; solid_box::get_weight, COMDAT
; _this$ = ecx
    push esi
    mov  esi, ecx
    push edi
    lea  ecx, DWORD PTR [esi+12]
    call ?get_density@solid_object@@QAEHXZ ; solid_object::get_density
    mov  ecx, esi
    mov  edi, eax
    call ?get_volume@box@@QAEHXZ ; box::get_volume
    imul eax, edi
    pop  edi
    pop  esi
    ret  0
?get_weight@solid_box@@QAEHXZ ENDP ; solid_box::get_weight
```

`get_weight()` appelle juste deux méthodes, mais pour `get_volume()` il passe simplement un pointeur sur `this`, et pour `get_density()` il passe un pointeur sur `this` incrémenté de 12 (ou `0xC`) octets, et ici, dans la disposition de la mémoire de la classe `solid_box`, les champs de la classe `solid_object` commencent.

Ainsi, la méthode `solid_object::get_density()` croira qu'elle traite une classe `solid_object` usuelle, et la méthode `box::get_volume()` fonctionnera avec ses trois champs, croyant que c'est juste un objet usuel de la classe box.

Ainsi, on peut dire, un objet d'une classe, qui hérite de plusieurs autres classes, est représenté en mémoire comme une classe *unifiée*, qui contient tous les champs hérités. Et chaque méthode héritée est appelée avec un pointeur sur la partie correspondante de la structure.

Méthodes virtuelles

Encode un exemple simple:

```
#include <stdio.h>
```

```
class object
{
    public:
        int color;
        object() { };
        object (int color) { this->color=color; };
        virtual void dump()
        {
            printf ("color=%d\n", color);
        };
};

class box : public object
{
    private:
        int width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is a box. color=%d, width=%d, height=%d, depth=%d⤦
    ↳ \n", color, width, height, depth);
        };
};

class sphere : public object
{
    private:
        int radius;
    public:
        sphere(int color, int radius)
        {
            this->color=color;
            this->radius=radius;
        };
        void dump()
        {
            printf ("this is sphere. color=%d, radius=%d\n", color, radius)⤦
    ↳ ;
        };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);
```

```
    object *o1=&b;
    object *o2=&s;

    o1->dump();
    o2->dump();
    return 0;
};
```

La classe *object* a une méthode virtuelle `dump()` qui est remplacée par celle de la classe héritant *box* et *sphere*.

Si nous sommes dans un environnement où le type de l'objet n'est pas connu, comme dans la fonction `main()` de l'exemple, où la méthode virtuelle `dump()` est appelée, l'information à propos de son type doit être stockée quelque part, afin d'être capable d'appeler la bonne méthode virtuelle.

Compilons-le dans MSVC 2008 avec les options `/Ox` et `/Ob0`, puis regardons le code de `main()` :

```
_s$ = -32 ; size = 12
_b$ = -20 ; size = 20
_main PROC
    sub  esp, 32
    push 30
    push 20
    push 10
    push 1
    lea  ecx, DWORD PTR _b$[esp+48]
    call ??0box@@QAE@HHHH@Z ; box::box
    push 40
    push 2
    lea  ecx, DWORD PTR _s$[esp+40]
    call ??0sphere@@QAE@HH@Z ; sphere::sphere
    mov  eax, DWORD PTR _b$[esp+32]
    mov  edx, DWORD PTR [eax]
    lea  ecx, DWORD PTR _b$[esp+32]
    call edx
    mov  eax, DWORD PTR _s$[esp+32]
    mov  edx, DWORD PTR [eax]
    lea  ecx, DWORD PTR _s$[esp+32]
    call edx
    xor  eax, eax
    add  esp, 32
    ret  0
_main ENDP
```

Un pointeur sur la fonction `dump()` est pris quelque part dans l'objet. Où pourrions-nous stocker l'adresse de la nouvelle méthode? Seulement quelque part dans le constructeur: il n'y a pas d'autre endroit puisque rien d'autre n'est appelé dans la fonction `main()`. [29]

[29]Vous pouvez en lire plus sur les pointeurs sur les fonctions dans la section afférente:(1.33 on page 495).

Regardons le code du constructeur de la classe *box* :

```
??_R0?AVbox@@@8 DD FLAT:??_7type_info@@6B@ ; box `RTTI Type Descriptor'
    DD    00H
    DB    '.?AVbox@@', 00H

??_R1A@?0A@EA@box@@8 DD FLAT:??_R0?AVbox@@@8 ; box::`RTTI Base Class
    Descriptor at (0,-1,0,64)'
    DD    01H
    DD    00H
    DD    0ffffffffH
    DD    00H
    DD    040H
    DD    FLAT:??_R3box@@8

??_R2box@@8 DD    FLAT:??_R1A@?0A@EA@box@@8 ; box::`RTTI Base Class Array'
    DD    FLAT:??_R1A@?0A@EA@object@@8

??_R3box@@8 DD    00H ; box::`RTTI Class Hierarchy Descriptor'
    DD    00H
    DD    02H
    DD    FLAT:??_R2box@@8

??_R4box@@6B@ DD 00H ; box::`RTTI Complete Object Locator'
    DD    00H
    DD    00H
    DD    FLAT:??_R0?AVbox@@@8
    DD    FLAT:??_R3box@@8

??_7box@@6B@ DD    FLAT:??_R4box@@6B@ ; box::`vftable'
    DD    FLAT:?dump@box@@UAEXXZ

_color$ = 8   ; size = 4
_width$ = 12  ; size = 4
_height$ = 16 ; size = 4
_depth$ = 20  ; size = 4
??0box@@QAE@HHHH@Z PROC ; box::box, COMDAT
; _this$ = ecx
    push esi
    mov  esi, ecx
    call ??0object@@QAE@XZ ; object::object
    mov  eax, DWORD PTR _color$[esp]
    mov  ecx, DWORD PTR _width$[esp]
    mov  edx, DWORD PTR _height$[esp]
    mov  DWORD PTR [esi+4], eax
    mov  eax, DWORD PTR _depth$[esp]
    mov  DWORD PTR [esi+16], eax
    mov  DWORD PTR [esi], OFFSET ??_7box@@6B@
    mov  DWORD PTR [esi+8], ecx
    mov  DWORD PTR [esi+12], edx
    mov  eax, esi
    pop  esi
    ret  16
??0box@@QAE@HHHH@Z ENDP ; box::box
```

Ici, nous avons une disposition de la mémoire légèrement différente: Le premier champ est un pointeur sur une table box::`vftable' (le nom a été mis par le compilateur MSVC).

Dans cette table nous voyons un lien sur une table nommée box::`RTTI Complete Object Locator' et aussi un lien sur la méthode box::dump().

Elles sont appelées table de méthodes virtuelles et RTTI[30]. La table de méthodes virtuelles a l'adresse des méthodes et la table RTTI contient l'information à propos des types.

À propos, les tables RTTI sont utilisées lors de l'appel à *dynamic_cast* et *typeid* en C++. Vous pouvez également voir ici le nom de la classe en chaîne de texte pur.

Ainsi, une méthode de la classe de base *object* peut aussi appelé la méthode virtuelle *object::dump()*, qui, en fait, va appeler une méthode d'une classe héritée, puisque cette information est présente juste dans la structure de l'objet.

Du temps CPU additionnel est requis pour faire la recherche dans ces tables et trouver l'adresse de la bonne méthode virtuelle, ainsi les méthodes virtuelles sont largement considérées comme légèrement plus lentes que les méthodes normales.

Dans le code généré par GCC, les tables RTTI sont construites légèrement différemment.

3.21.2 ostream

Recommençons avec l'exemple «hello world», mais cette fois nous allons utiliser *ostream* :

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

Presque tous les livres sur C++ nous disent que l'opérateur << peut-être défini (*surchargé*) pour tous les types. C'est ce qui est fait dans *ostream*. Nous voyons que operator<< est appelé pour *ostream* :

Listing 3.101 : MSVC 2012 (listing réduit)

```
$SG37112 DB 'Hello, world!', 0aH, 00H

_main PROC
    push OFFSET $SG37112
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ;
    std::cout
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?↙
    ↘ $char_traits@D@std@@@0@AAV10@PBD@Z ;
    std::operator<<<std::char_traits<char> >
```

[30]Run-Time Type Information

```
    add  esp, 8
    xor  eax, eax
    ret  0
_main ENDP
```

Modifions l'exemple:

```
#include <iostream>

int main()
{
    std::cout << "Hello, " << "world!\n";
}
```

À nouveau, dans chaque livre sur C++ nous lisons que le résultat de chaque `operator<<` dans ostream est transmis au suivant. En effet:

Listing 3.102 : MSVC 2012

```
$SG37112 DB 'world!', 0aH, 00H
$SG37113 DB 'Hello, ', 00H

_main PROC
    push OFFSET $SG37113 ; 'Hello, '
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ;
    std::cout
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?↙
    ↘ $char_traits@D@std@@@0@AAV10@PBD@Z ;
    std::operator<<<std::char_traits<char> >
    add  esp, 8

    push OFFSET $SG37112 ; 'world!'
    push eax             ; résultat de l'exécution précédente de la fonction
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?↙
    ↘ $char_traits@D@std@@@0@AAV10@PBD@Z ;
    std::operator<<<std::char_traits<char> >
    add  esp, 8

    xor  eax, eax
    ret  0
_main ENDP
```

Si nous renommions la méthode `operator<<` en `f()`, ce code ressemblerait à ceci:

```
f(f(std::cout, "Hello, "), "world!");
```

GCC génère presque le même code que MSVC.

3.21.3 Références

En C++, les références sont aussi des pointeurs (3.23 on page 773), mais elles sont dites *sûre*, car il est plus difficile de faire une erreur en les utilisant (C++11 8.3.2).

Par exemple, les références doivent toujours pointer sur un objet de type correspondant et ne peuvent pas être NULL [Marshall Cline, *C++ FAQ*8.6].

Encore mieux que ça, les références ne peuvent être changées, il est impossible de les faire pointer sur un autre objet (réassigner) [Marshall Cline, *C++ FAQ*8.5].

Si nous essayons de modifier l'exemple avec des pointeurs (3.23 on page 773) pour utiliser des références à la place …

```
void f2 (int x, int y, int & sum, int & product)
{
        sum=x+y;
        product=x*y;
};
```

…alors nous pouvons voir que le code compilé est simplement le même que dans l'exemple avec les pointeurs (3.23 on page 773) :

Listing 3.103 : MSVC 2010 avec optimisation

```
_x$ = 8         ; size = 4
_y$ = 12        ; size = 4
_sum$ = 16      ; size = 4
_product$ = 20  ; size = 4
?f2@@YAXHHAAH0@Z PROC   ; f2
        mov     ecx, DWORD PTR _y$[esp-4]
        mov     eax, DWORD PTR _x$[esp-4]
        lea     edx, DWORD PTR [eax+ecx]
        imul eax, ecx
        mov ecx, DWORD PTR _product$[esp-4]
        push esi
        mov     esi, DWORD PTR _sum$[esp]
        mov     DWORD PTR [esi], edx
        mov     DWORD PTR [ecx], eax
        pop     esi
        ret     0
?f2@@YAXHHAAH0@Z ENDP   ; f2
```

(La raison pour laquelle les fonctions C++ ont des noms aussi étranges est expliquée ici: 3.21.1 on page 699.)

De ce fait, les références C++ sont bien plus efficientes que les pointeurs usuels.

3.21.4 STL

N.B.: tous les exemples ici ont été testés uniquement en environnement 32-bit. x64 non testé.

std::string

Internals

De nombreuses bibliothèques de chaînes [Dennis Yurichev, *C/C++ programming language notes*2.2] implémentent une structure qui contient un pointeur sur un buffer de chaîne, une variable qui contient toujours la longueur actuelle de la chaîne (ce qui est très pratique pour de nombreuses fonctions: [Dennis Yurichev, *C/C++ programming language notes*2.2.1]) et une variable qui contient la taille actuelle du buffer.

La chaîne dans le buffer est en général terminée par un zéro, afin de pouvoir passer un pointeur sur le buffer aux fonctions qui prennent une chaîne C ASCIIZ standard.

Il n'est pas précisé dans le standard C++ comment std::string doit être implémentée, toutefois, elle l'est en général comme expliqué ci-dessus.

La chaîne C++ n'est pas une classse (comme QString dansQt, par exemple) mais un template (basic_string), ceci est fait afin de supporter différents types de caractères: au moins *char* et *wchar_t*.

Donc, std::string est une classe avec *char* comme type de base.

Et std::wstring est une classe avec *wchar_t* comme type de base.

MSVC

L'implémentation de MSVC peut stocker le buffer en place au lieu d'utiliser un pointeur sur un buffer (si la chaîne est plus courte que 16 symboles).

Ceci implique qu'une chaîne courte occupe au moins $16+4+4=24$ octets en environnement 32-bit et au moins $16+8+8=32$ octets dans un 64-bit, et si la chaîne est plus longue que 16 caractères, nous devons ajouter la longueur de la chaîne elle-même.

Listing 3.104 : exemple pour MSVC

```
#include <string>
#include <stdio.h>

struct std_string
{
    union
    {
        char buf[16];
        char* ptr;
    } u;
    size_t size;     // AKA 'Mysize' dans MSVC
    size_t capacity; // AKA 'Myres' dans MSVC
};

void dump_std_string(std::string s)
{
    struct std_string *p=(struct std_string*)&s;
    printf ("[%s] size:%d capacity:%d\n", p->size>16 ? p->u.ptr : p->u.buf,
    ↪ p->size, p->capacity);
};

int main()
```

```
{
    std::string s1="a short string";
    std::string s2="a string longer than 16 bytes";

    dump_std_string(s1);
    dump_std_string(s2);

    // cela fonctionne sans utiliser c_str()
    printf ("%s\n", &s1);
    printf ("%s\n", s2);
};
```

Presque tout est clair dans le code source.

Quelques notes:

Si la chaîne est plus petite que 16 symboles, il n'y a pas de buffer alloué sur le tas.

Ceci est pratique car en pratique, beaucoup de chaînes sont courtes.

Il semble que les développeurs de Microsoft aient choisi 16 caractères comme un bon compromis.

On voit une chose très importante à la fin de main() : nous n'utilisons pas la méthode c_str(), néanmoins, si nous compilons et exécutons ce code, les deux chaînes apparaîtrons dans la console!

C'est pourquoi ceci fonctionne.

Dans le premier cas, la chaîne fait moins de 16 caractères et le buffer avec la chaîne se trouve au début de l'objet std::string (il peut-être traité comme une structure). printf() traite le pointeur comme un pointeur sur un tableau de caractères terminé par un
0, donc ça fonctionne.

L'affichage de la seconde chaîne (de plus de 16 caractères) est encore plus dangereux: c'est une erreur typique de programmeur (ou une typo) d'oublier d'écrire c_str().

Ceci fonctionne car pour le moment un pointeur sur le buffer est situé au début de la structure.

Ceci peut passer inaperçu pendant un long moment, jusqu'à ce qu'une chaîne plus longue apparaisse à un moment, alors le processus plantera.

GCC

L'implémentation de GCC de cette structure a une variable de plus—le compteur de références.

Une chose intéressante est que dans GCC un pointeur sur une instance de std::string ne pointe pas au début de la structure, mais sur le pointeur du buffer. Dans *libstdc++-v3\include\bits\basic_string.h* nous pouvons lire que ça a été fait ainsi afin de faciliter le débogage:

```
 *  The reason you want _M_data pointing to the character %array and
 *  not the _Rep is so that the debugger can see the string
 *  contents. (Probably we should add a non-inline member to get
 *  the _Rep for the debugger to use, so users can check the actual
 *  string length.)
```

code source de basic_string.h

Considérons ceci dans notre exemple:

Listing 3.105 : exemple pour GCC

```
#include <string>
#include <stdio.h>

struct std_string
{
    size_t length;
    size_t capacity;
    size_t refcount;
};

void dump_std_string(std::string s)
{
    char *p1=*(char**)&s; // GCC type checking workaround
    struct std_string *p2=(struct std_string*)(p1-sizeof(struct std_string)
    ↪ );
    printf ("[%s] size:%d capacity:%d\n", p1, p2->length, p2->capacity);
};

int main()
{
    std::string s1="a short string";
    std::string s2="a string longer than 16 bytes";

    dump_std_string(s1);
    dump_std_string(s2);

    // GCC type checking workaround:
    printf ("%s\n", *(char**)&s1);
    printf ("%s\n", *(char**)&s2);
};
```

Il faut utiliser une astuce pour imiter l’erreur que nous avons vue avant car GCC a une vérification de type plus forte, cependant, printf() fonctionne ici également sans c_str().

Un exemple plus avancé

```
#include <string>
#include <stdio.h>
```

```
int main()
{
    std::string s1="Hello, ";
    std::string s2="world!\n";
    std::string s3=s1+s2;

    printf ("%s\n", s3.c_str());
}
```

Listing 3.106 : MSVC 2012

```
$SG39512 DB 'Hello, ', 00H
$SG39514 DB 'world!', 0aH, 00H
$SG39581 DB '%s', 0aH, 00H

_s2$ = -72 ; size = 24
_s3$ = -48 ; size = 24
_s1$ = -24 ; size = 24
_main PROC
    sub  esp, 72

    push 7
    push OFFSET $SG39512
    lea  ecx, DWORD PTR _s1$[esp+80]
    mov  DWORD PTR _s1$[esp+100], 15
    mov  DWORD PTR _s1$[esp+96], 0
    mov  BYTE PTR _s1$[esp+80], 0
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?↙
    ↳ $allocator@D@2@@std@@QAEAAV12@PBDI@Z ;
    std::basic_string<char,std::char_traits<char>,std::allocator<char> >::assign

    push 7
    push OFFSET $SG39514
    lea  ecx, DWORD PTR _s2$[esp+80]
    mov  DWORD PTR _s2$[esp+100], 15
    mov  DWORD PTR _s2$[esp+96], 0
    mov  BYTE PTR _s2$[esp+80], 0
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?↙
    ↳ $allocator@D@2@@std@@QAEAAV12@PBDI@Z ;
    std::basic_string<char,std::char_traits<char>,std::allocator<char> >::assign

    lea  eax, DWORD PTR _s2$[esp+72]
    push eax
    lea  eax, DWORD PTR _s1$[esp+76]
    push eax
    lea  eax, DWORD PTR _s3$[esp+80]
    push eax
    call ??$?HDU?$char_traits@D@std@@V?$allocator@D@1@@std@@YA?AV?↙
    ↳ $basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@0@ABV10@0@Z ;
    std::operator+<char,std::char_traits<char>,std::allocator<char> >

    ; méthode c_str() mise en ligne (inlined):
    cmp  DWORD PTR _s3$[esp+104], 16
    lea  eax, DWORD PTR _s3$[esp+84]
```

```
    cmovae eax, DWORD PTR _s3$[esp+84]

    push eax
    push OFFSET $SG39581
    call _printf
    add  esp, 20

    cmp  DWORD PTR _s3$[esp+92], 16
    jb   SHORT $LN119@main
    push DWORD PTR _s3$[esp+72]
    call ??3@YAXPAX@Z                ; opérateur delete
    add  esp, 4
$LN119@main:
    cmp  DWORD PTR _s2$[esp+92], 16
    mov  DWORD PTR _s3$[esp+92], 15
    mov  DWORD PTR _s3$[esp+88], 0
    mov  BYTE PTR _s3$[esp+72], 0
    jb   SHORT $LN151@main
    push DWORD PTR _s2$[esp+72]
    call ??3@YAXPAX@Z                ; opérateur delete
    add  esp, 4
$LN151@main:
    cmp  DWORD PTR _s1$[esp+92], 16
    mov  DWORD PTR _s2$[esp+92], 15
    mov  DWORD PTR _s2$[esp+88], 0
    mov  BYTE PTR _s2$[esp+72], 0
    jb   SHORT $LN195@main
    push DWORD PTR _s1$[esp+72]
    call ??3@YAXPAX@Z                ; opérateur delete
    add  esp, 4
$LN195@main:
    xor  eax, eax
    add  esp, 72
    ret  0
_main ENDP
```

Le compilateur ne construit pas les chaînes statiquement: il ne serait de toutes façons pas possible si le buffer devait être situer dans le tas.

au lieu de ça, les chaînes ASCIIZ sont stockées dans le segment de données, et plus tard, au lancement, avec l'aide de la méthode «assign», les chaînes s1 et s2 sont construites.

Veuillez noter qu'il n'y a pas d'appel à la méthode c_str(), car le code est assez petit pour que le compilateur le mette en ligne ici: si la chaîne est plus courte que 16 caractères, un pointeur sur le buffer est laissé dans EAX, autrement l'adresse du buffer de la chaîne située dans le tas est récupérée.

Ensuite, nous voyons les appels aux 3 destructeurs, is sont appelés si la chaîne fait plus de 16 caractères: le buffer dans le tas doit être libéré. Autrement, puisque les trois objets std::string sont stockés dans la pile, ils sont automatiquement libérés lorsque la fonction se termine.

En conséquence, traiter des chaînes courtes est plus rapide, car il y a moins d'accès

au tas.

Le code de GCC est encore plus simple (car la manière de GCC, comme nous l'avons vu ci-dessus, est de ne pas stocker les chaînes courtes directement dans la structure) :

Listing 3.107 : GCC 4.8.1

```
.LC0:
    .string "Hello, "
.LC1:
    .string "world!\n"
main:
    push ebp
    mov  ebp, esp
    push edi
    push esi
    push ebx
    and  esp, -16
    sub  esp, 32
    lea  ebx, [esp+28]
    lea  edi, [esp+20]
    mov  DWORD PTR [esp+8], ebx
    lea  esi, [esp+24]
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC0
    mov  DWORD PTR [esp], edi

    call _ZNSsC1EPKcRKSaIcE

    mov  DWORD PTR [esp+8], ebx
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC1
    mov  DWORD PTR [esp], esi

    call _ZNSsC1EPKcRKSaIcE

    mov  DWORD PTR [esp+4], edi
    mov  DWORD PTR [esp], ebx

    call _ZNSsC1ERKSs

    mov  DWORD PTR [esp+4], esi
    mov  DWORD PTR [esp], ebx

    call _ZNSs6appendERKSs

    ; c_str() mis en ligne (inlined):
    mov  eax, DWORD PTR [esp+28]
    mov  DWORD PTR [esp], eax

    call puts

    mov  eax, DWORD PTR [esp+28]
    lea  ebx, [esp+19]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
```

```
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    mov  eax, DWORD PTR [esp+24]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    mov  eax, DWORD PTR [esp+20]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    lea  esp, [ebp-12]
    xor  eax, eax
    pop  ebx
    pop  esi
    pop  edi
    pop  ebp
    ret
```

On voit que ce n'est pas un pointeur sur l'objet qui est passé aux destructeurs, mais plutôt une adresse 12 octets (ou 3 mots) avant, i.e., un pointeur sur le début réel de la structure.

std::string comme une variable globale

Les programmeurs C++ expérimentés savent que les des variables globales des types STL[31] peuvent être définis sans problème.

Oui, en effet:

```
#include <stdio.h>
#include <string>

std::string s="a string";

int main()
{
    printf ("%s\n", s.c_str());
};
```

Mais comment et où le constructeur de `std::string` sera appelé?

En fait, cette variable sera initialisée avant le démarrage de `main()`.

Listing 3.108 : MSVC 2012: voici comment une variable globale est construite et aussi comment sont destructeur est déclaré

```
??__Es@@YAXXZ PROC
    push 8
    push OFFSET $SG39512 ; 'a string'
```

[31](C++) Standard Template Library

```
    mov  ecx, OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↙
    ↘ $allocator@D@2@@std@@A ;
    s
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?↙
    ↘ $allocator@D@2@@std@@QAEAAV12@PBDI@Z ;
    std::basic_string<char,std::char_traits<char>,std::allocator<char> >::assign
    push OFFSET ??__Fs@@YAXXZ ; `dynamic atexit destructor for 's''
    call _atexit
    pop  ecx
    ret  0
??__Es@@YAXXZ ENDP
```

Listing 3.109 : MSVC 2012: ici une variable globale est utilisée dans main()

```
$SG39512 DB 'a string', 00H
$SG39519 DB '%s', 0aH, 00H

_main PROC
    cmp  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↙
    ↘ $allocator@D@2@@std@@A+20, 16
    mov  eax, OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↙
    ↘ $allocator@D@2@@std@@A ;
    s
    cmovae eax, DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↙
    ↘ $allocator@D@2@@std@@A
    push eax
    push OFFSET $SG39519 ; '%s'
    call _printf
    add  esp, 8
    xor  eax, eax
    ret  0
_main ENDP
```

Listing 3.110 : MSVC 2012: cette fonction destructeur est appelée avant exit

```
??__Fs@@YAXXZ PROC
    push ecx
    cmp  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↙
    ↘ $allocator@D@2@@std@@A+20, 16
    jb   SHORT $LN23@dynamic
    push esi
    mov  esi, DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↙
    ↘ $allocator@D@2@@std@@A
    lea  ecx, DWORD PTR $T2[esp+8]
    call ??0?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ
    push OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↙
    ↘ $allocator@D@2@@std@@A ;
    s
    lea  ecx, DWORD PTR $T2[esp+12]
    call ??$destroy@PAD@?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAEXPAPAD@Z
    lea  ecx, DWORD PTR $T1[esp+8]
    call ??0?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ
    push esi
    call ??3@YAXPAX@Z ; operator delete
    add  esp, 4
```

```
    pop  esi
$LN23@dynamic:
    mov  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↙
    ↘ $allocator@D@2@@std@@A+20, 15
    mov  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↙
    ↘ $allocator@D@2@@std@@A+16, 0
    mov  BYTE PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↙
    ↘ $allocator@D@2@@std@@A, 0
    pop  ecx
    ret  0
??__Fs@@YAXXZ ENDP
```

En fait, une fonction spéciale avec tous les constructeurs des variables globales est appelée depuis CRT, avant main().

Mieux que ça: avec l'aide de atexit() une autre fonction est enregistrée, qui contient des appels aux destructeurs de telles variables globales.

GCC fonctionne comme ceci:

Listing 3.111 : GCC 4.8.1

```
main:
    push ebp
    mov  ebp, esp
    and  esp, -16
    sub  esp, 16
    mov  eax, DWORD PTR s
    mov  DWORD PTR [esp], eax
    call puts
    xor  eax, eax
    leave
    ret
.LC0:
    .string "a string"
_GLOBAL__sub_I_s:
    sub  esp, 44
    lea  eax, [esp+31]
    mov  DWORD PTR [esp+8], eax
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC0
    mov  DWORD PTR [esp], OFFSET FLAT:s
    call _ZNSsC1EPKcRKSaIcE
    mov  DWORD PTR [esp+8], OFFSET FLAT:__dso_handle
    mov  DWORD PTR [esp+4], OFFSET FLAT:s
    mov  DWORD PTR [esp], OFFSET FLAT:_ZNSsD1Ev
    call __cxa_atexit
    add  esp, 44
    ret
.LFE645:
    .size  _GLOBAL__sub_I_s, .-_GLOBAL__sub_I_s
    .section .init_array,"aw"
    .align 4
    .long  _GLOBAL__sub_I_s
    .globl s
```

```
    .bss
    .align 4
    .type  s, @object
    .size  s, 4
s:
    .zero  4
    .hidden __dso_handle
```

Mais il ne crée pas une fonction séparée pour cela, chaque destructeur est passé à atexit(), un par un.

std::list

Ceci est la célèbre liste doublement chaînée: chaque élément a deux pointeurs, un sur l'élément précédent et un sur le suivant.

Ceci implique que l'empreinte mémoire est augmentés de 2 mots pour chaque élément (8 octets dans un environnement 32-bit ou 16v octets en 64-bit).

C++ STL ajoute juste les pointeurs «next» et «previous» à la structure existante du type que vous voulez unir dans une liste.

Travaillons sur un exemple avec une simple structure de deux variables que nous voulons stocker dans une liste.

Bien que le standard C++ ne dise pas comme l'implémenter, GCC et MSVC l'implémentent de manière directe et similaire, donc il n'y a qu'un seul code source pour les deux:

```
#include <stdio.h>
#include <list>
#include <iostream>

struct a
{
    int x;
    int y;
};

struct List_node
{
    struct List_node* _Next;
    struct List_node* _Prev;
    int x;
    int y;
};

void dump_List_node (struct List_node *n)
{
    printf ("ptr=0x%p _Next=0x%p _Prev=0x%p x=%d y=%d\n",
        n, n->_Next, n->_Prev, n->x, n->y);
};

void dump_List_vals (struct List_node* n)
```

```
{
    struct List_node* current=n;

    for (;;)
    {
        dump_List_node (current);
        current=current->_Next;
        if (current==n) // end
            break;
    };
};

void dump_List_val (unsigned int *a)
{
#ifdef _MSC_VER
    // l'implémentation de GCC n'a pas le champ "size"
    printf ("_Myhead=0x%p, _Mysize=%d\n", a[0], a[1]);
#endif
    dump_List_vals ((struct List_node*)a[0]);
};

int main()
{
    std::list<struct a> l;

    printf ("* empty list:\n");
    dump_List_val((unsigned int*)(void*)&l);

    struct a t1;
    t1.x=1;
    t1.y=2;
    l.push_front (t1);
    t1.x=3;
    t1.y=4;
    l.push_front (t1);
    t1.x=5;
    t1.y=6;
    l.push_back (t1);

    printf ("* 3-elements list:\n");
    dump_List_val((unsigned int*)(void*)&l);

    std::list<struct a>::iterator tmp;
    printf ("node at .begin:\n");
    tmp=l.begin();
    dump_List_node ((struct List_node *)*(void**)&tmp);
    printf ("node at .end:\n");
    tmp=l.end();
    dump_List_node ((struct List_node *)*(void**)&tmp);

    printf ("* let's count from the beginning:\n");
    std::list<struct a>::iterator it=l.begin();
    printf ("1st element: %d %d\n", (*it).x, (*it).y);
```

```
    it++;
    printf ("2nd element: %d %d\n", (*it).x, (*it).y);
    it++;
    printf ("3rd element: %d %d\n", (*it).x, (*it).y);
    it++;
    printf ("element at .end(): %d %d\n", (*it).x, (*it).y);

    printf ("* let's count from the end:\n");
    std::list<struct a>::iterator it2=l.end();
    printf ("element at .end(): %d %d\n", (*it2).x, (*it2).y);
    it2--;
    printf ("3rd element: %d %d\n", (*it2).x, (*it2).y);
    it2--;
    printf ("2nd element: %d %d\n", (*it2).x, (*it2).y);
    it2--;
    printf ("1st element: %d %d\n", (*it2).x, (*it2).y);

    printf ("removing last element...\n");
    l.pop_back();
    dump_List_val((unsigned int*)(void*)&l);
};
```

GCC

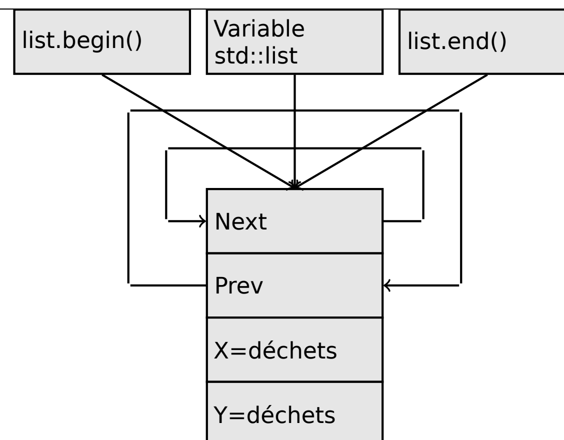
Commençons avec GCC.

Lorsque nous lançons l'exemple, nous voyons un long dump, travaillons avec par morceaux.

```
* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
```

Ici, nous voyons une liste vide.

Bien que ce soit une liste vide, elle a un élément avec des données non initialisées (AKA *dummy node*) dans x et y. Les deux pointeurs «next» et «prev» pointeurs sur le nœud lui-même:



À ce moment, les itérateurs .begin et .end sont égaux l'un à l'autre.

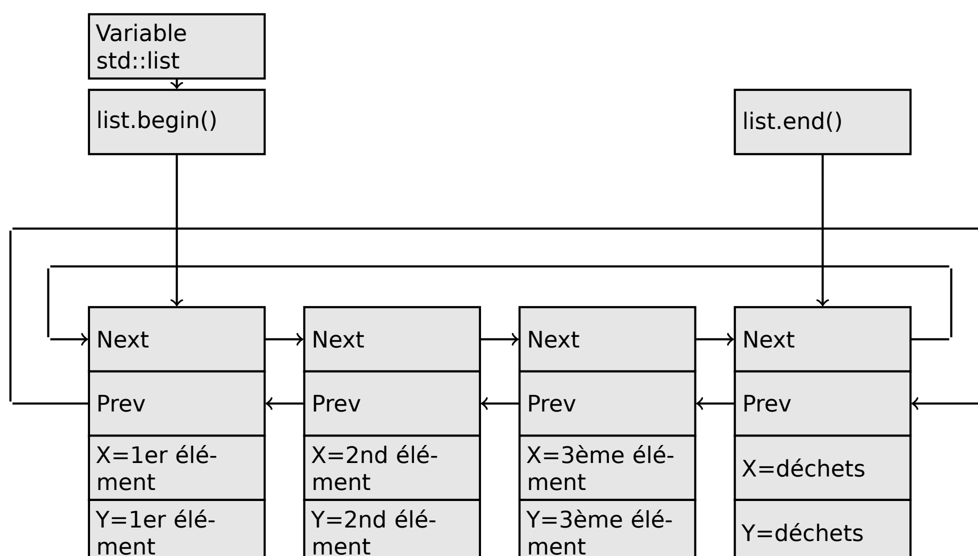
Si nous poussons 3 éléments, la liste interne sera:

```
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

Le dernier élément est encore en 0x0028fe90, il ne sera pas déplacé avant l'élimination de la liste.

Il contient toujours des valeurs aléatoires dans x et y (5 et 6). Par coïncidence, ces valeurs sont les même que dans le dernier élément, mais ça ne signifie pas que ça soit significatif.

Voici comment ces 3 éléments sont stockés en mémoire:



La variable l pointe toujours sur le premier nœud.

Les itérateurs .begin() et .end() ne sont pas des variables, mais des fonctions, qui renvoient des pointeurs sur le nœud correspondant lorsqu'elle sont appelées.

Avoir un élément fictif (AKA *nœud sentinelle*) est une pratique répandue dans l'implémentation des listes doublements chaînées.

Sans lui, de nombreuses opérations deviennent légèrement plus complexes et de ce fait, plus lentes.

L'itérateur est en fait juste un pointeur sur un nœud. list.begin() et list.end() retourne juste des pointeurs.

```
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

Le fait que le dernier élément ait un pointeur sur le premier et le premier élément ait un pointeur sur le dernier nous rappelle les listes circulaires.

Ceci est très pratique ici: en ayant un pointeur sur le premier élément de la liste, i.e., ce qui est dans la variable l, il est très facile d'obtenir rapidement un pointeur sur le dernier, sans devoir traverser toute la liste.

Insérer un élément à la fin de la liste est également rapide, grâce à cette caractéristique.

`operator--` et `operator++` mettent la valeur courante de l'itérateur à la valeur `current_node->prev` ou `current_node->next`.

Les itérateurs inverses (.rbegin, .rend) fonctionne de la même façon, mais à l'envers.

`operator*` renvoie un pointeur sur le point dans la structure du nœud, où la structure débute, i.e., un pointeur sur le premier élément de la structure (x).

L'insertion et la suppression dans la liste sont triviaux: simplement allouer un nouveau nœud (ou le désallouer) et mettre à jour les pointeurs afin qu'ils soient valides.

C'est pourquoi un itérateur peut devenir invalide après la suppression d'un élément: il peut toujours pointer sur le nœud qui a été déjà désalloué. Ceci est aussi appelé un *dangling pointer*.

Et bien sûr, l'information sur le nœud libéré (sur lequel pointe toujours l'itérateur) ne peut plus être utilisée.

L'implémentation de GCC (à partir de 4.8.1) ne stocke plus la taille courante de la liste: ceci implique une méthode .size() lente: il doit traverser toute la liste pour compter les éléments, car il n'a pas d'autre moyen d'obtenir l'information.

Ceci signifie que cette opération est en $O(n)$, i.e., elle devient constamment plus lente lorsque la liste grandit.

Listing 3.112 : GCC 4.8.1 -fno-inline-small-functions avec optimisation

```
main proc near
    push ebp
```

```
    mov  ebp, esp
    push esi
    push ebx
    and  esp, 0FFFFFFF0h
    sub  esp, 20h
    lea  ebx, [esp+10h]
    mov  dword ptr [esp], offset s ; "* empty list:"
    mov  [esp+10h], ebx
    mov  [esp+14h], ebx
    call puts
    mov  [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    lea  esi, [esp+18h]
    mov  [esp+4], esi
    mov  [esp], ebx
    mov  dword ptr [esp+18h], 1 ; X pour le nouvel élément
    mov  dword ptr [esp+1Ch], 2 ; Y pour le nouvel élément
    call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ;
    std::list<a,std::allocator<a>>::push_front(a const&)
    mov  [esp+4], esi
    mov  [esp], ebx
    mov  dword ptr [esp+18h], 3 ; X pour le nouvel élément
    mov  dword ptr [esp+1Ch], 4 ; Y pour le nouvel élément
    call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ;
    std::list<a,std::allocator<a>>::push_front(a const&)
    mov  dword ptr [esp], 10h
    mov  dword ptr [esp+18h], 5 ; X pour le nouvel élément
    mov  dword ptr [esp+1Ch], 6 ; Y pour le nouvel élément
    call _Znwj           ; opérateur new(uint)
    cmp  eax, 0FFFFFFF8h
    jz   short loc_80002A6
    mov  ecx, [esp+1Ch]
    mov  edx, [esp+18h]
    mov  [eax+0Ch], ecx
    mov  [eax+8], edx

loc_80002A6: ; CODE XREF: main+86
    mov  [esp+4], ebx
    mov  [esp], eax
    call _ZNSt8__detail15_List_node_base7_M_hookEPS0_ ;
    std::__detail::_List_node_base::_M_hook(std::__detail::_List_node_base*)
    mov  dword ptr [esp], offset a3ElementsList ; "* 3-elements list:"
    call puts
    mov  [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    mov  dword ptr [esp], offset aNodeAt_begin ; "node at .begin:"
    call puts
    mov  eax, [esp+10h]
    mov  [esp], eax
    call _Z14dump_List_nodeP9List_node ; dump_List_node(List_node *)
    mov  dword ptr [esp], offset aNodeAt_end ; "node at .end:"
    call puts
    mov  [esp], ebx
    call _Z14dump_List_nodeP9List_node ; dump_List_node(List_node *)
```

```
mov  dword ptr [esp], offset aLetSCountFromT ; "* let's count from the
beginning:"
call puts
mov  esi, [esp+10h]
mov  eax, [esi+0Ch]
mov  [esp+0Ch], eax
mov  eax, [esi+8]
mov  dword ptr [esp+4], offset a1stElementDD ; "1st element: %d %d\n"
mov  dword ptr [esp], 1
mov  [esp+8], eax
call __printf_chk
mov  esi, [esi] ; operator++: get ->next pointer
mov  eax, [esi+0Ch]
mov  [esp+0Ch], eax
mov  eax, [esi+8]
mov  dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
mov  dword ptr [esp], 1
mov  [esp+8], eax
call __printf_chk
mov  esi, [esi] ; operator++: get ->next pointer
mov  eax, [esi+0Ch]
mov  [esp+0Ch], eax
mov  eax, [esi+8]
mov  dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
mov  dword ptr [esp], 1
mov  [esp+8], eax
call __printf_chk
mov  eax, [esi] ; operator++: get ->next pointer
mov  edx, [eax+0Ch]
mov  [esp+0Ch], edx
mov  eax, [eax+8]
mov  dword ptr [esp+4], offset aElementAt_endD ;
"element at .end(): %d %d\n"
mov  dword ptr [esp], 1
mov  [esp+8], eax
call __printf_chk
mov  dword ptr [esp], offset aLetSCountFro_0 ; "* let's count from the
end:"
call puts
mov  eax, [esp+1Ch]
mov  dword ptr [esp+4], offset aElementAt_endD ;
"element at .end(): %d %d\n"
mov  dword ptr [esp], 1
mov  [esp+0Ch], eax
mov  eax, [esp+18h]
mov  [esp+8], eax
call __printf_chk
mov  esi, [esp+14h]
mov  eax, [esi+0Ch]
mov  [esp+0Ch], eax
mov  eax, [esi+8]
mov  dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
mov  dword ptr [esp], 1
mov  [esp+8], eax
```

```
    call __printf_chk
    mov  esi, [esi+4] ; operator--: get ->prev pointer
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  eax, [esi+4] ; operator--: get ->prev pointer
    mov  edx, [eax+0Ch]
    mov  [esp+0Ch], edx
    mov  eax, [eax+8]
    mov  dword ptr [esp+4], offset a1stElementDD ; "1st element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  dword ptr [esp], offset aRemovingLastEl ; "removing last element..."
    call puts
    mov  esi, [esp+14h]
    mov  [esp], esi
    call _ZNSt8__detail15_List_node_base9_M_unhookEv ;
    std::__detail::_List_node_base::_M_unhook(void)
    mov  [esp], esi ; void *
    call _ZdlPv ; operator delete(void *)
    mov  [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    mov  [esp], ebx
    call _ZNSt10_List_baseI1aSaIS0_EE8_M_clearEv ;
    std::_List_base<a,std::allocator<a>>::_M_clear(void)
    lea  esp, [ebp-8]
    xor  eax, eax
    pop  ebx
    pop  esi
    pop  ebp
    retn
main endp
```

Listing 3.113 : La sortie complète

```
* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
* let's count from the beginning:
1st element: 3 4
2nd element: 1 2
```

```
3rd element: 5 6
element at .end(): 5 6
* let's count from the end:
element at .end(): 5 6
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x0028fe90 _Prev=0x000349a0 x=1 y=2
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034988 x=5 y=6
```

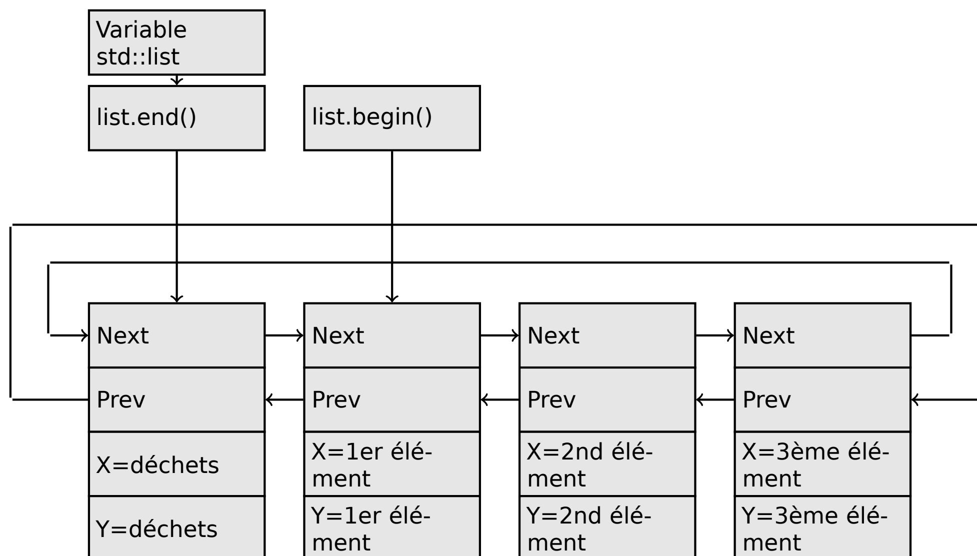
MSVC

L'implémentation de MSVC (2012) est la même, mais elle stocke aussi la taille courante de la liste.

Ceci implique que la méthode .size() est très rapide ($O(1)$) : elle doit juste lire une valeur depuis la mémoire.

D'un autre côté, la variable size doit être mise à jour à chaque insertion/suppression.

L'implémentation de MSVC est aussi légèrement différente dans la façon dont elle arrange les nœuds:



GCC a son élément fictif à la fin de la liste, tandis que MSVC l'a au début.

Listing 3.114 : MSVC 2012 avec optimisation /Fa2.asm /GS- /Ob1

```
_l$ = -16 ; size = 8
_t1$ = -8 ; size = 8
_main    PROC
    sub  esp, 16
    push ebx
```

```
push esi
push edi
push 0
push 0
lea  ecx, DWORD PTR _l$[esp+36]
mov  DWORD PTR _l$[esp+40], 0
; allocate first garbage element
call ?_Buynode0@?$_List_alloc@$0A@U?$_List_base_types@Ua@@V?↙
↘ $allocator@Ua@@@std@@@std@@@std@@QAEPAU?↙
↘ $_List_node@Ua@@PAX@2@PAU32@0@Z ;
std::_List_alloc<0,std::_List_base_types<a,std::allocator<a> > >::_Buynode0
mov  edi, DWORD PTR __imp__printf
mov  ebx, eax
push OFFSET $SG40685 ; '* empty list:'
mov  DWORD PTR _l$[esp+32], ebx
call edi  ; printf
lea  eax, DWORD PTR _l$[esp+32]
push eax
call ?dump_List_val@@YAXPAI@Z ; dump_List_val
mov  esi, DWORD PTR [ebx]
add  esp, 8
lea  eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [esi+4]
lea  ecx, DWORD PTR _l$[esp+36]
push esi
mov  DWORD PTR _t1$[esp+40], 1 ; data for a new node
mov  DWORD PTR _t1$[esp+44], 2 ; data for a new node
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?↙
↘ $allocator@Ua@@@std@@@std@@QAEPAU?↙
↘ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ;
std::_List_buy<a,std::allocator<a> >::_Buynode<a const &>
mov  DWORD PTR [esi+4], eax
mov  ecx, DWORD PTR [eax+4]
mov  DWORD PTR _t1$[esp+28], 3 ; data for a new node
mov  DWORD PTR [ecx], eax
mov  esi, DWORD PTR [ebx]
lea  eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [esi+4]
lea  ecx, DWORD PTR _l$[esp+36]
push esi
mov  DWORD PTR _t1$[esp+44], 4 ; data for a new node
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?↙
↘ $allocator@Ua@@@std@@@std@@QAEPAU?↙
↘ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ;
std::_List_buy<a,std::allocator<a> >::_Buynode<a const &>
mov  DWORD PTR [esi+4], eax
mov  ecx, DWORD PTR [eax+4]
mov  DWORD PTR _t1$[esp+28], 5 ; data for a new node
mov  DWORD PTR [ecx], eax
lea  eax, DWORD PTR _t1$[esp+28]
```

```
push eax
push DWORD PTR [ebx+4]
lea  ecx, DWORD PTR _l$[esp+36]
push ebx
mov  DWORD PTR _t1$[esp+44], 6 ; data for a new node
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?↙
↳ $allocator@Ua@@@std@@@std@@QAEPAU?↙
↳ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ;
std::_List_buy<a,std::allocator<a> >::_Buynode<a const &>
mov  DWORD PTR [ebx+4], eax
mov  ecx, DWORD PTR [eax+4]
push OFFSET $SG40689 ; '* 3-elements list:'
mov  DWORD PTR _l$[esp+36], 3
mov  DWORD PTR [ecx], eax
call edi ; printf
lea  eax, DWORD PTR _l$[esp+32]
push eax
call ?dump_List_val@@YAXPAI@Z ; dump_List_val
push OFFSET $SG40831 ; 'node at .begin:'
call edi ; printf
push DWORD PTR [ebx] ; get next field of node "l" variable points to
call ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
push OFFSET $SG40835 ; 'node at .end:'
call edi ; printf
push ebx ; pointer to the node "l" variable points to!
call ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
push OFFSET $SG40839 ; '* let''s count from the begin:'
call edi ; printf
mov  esi, DWORD PTR [ebx] ; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40846 ; '1st element: %d %d'
call edi ; printf
mov  esi, DWORD PTR [esi] ; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40848 ; '2nd element: %d %d'
call edi ; printf
mov  esi, DWORD PTR [esi] ; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40850 ; '3rd element: %d %d'
call edi ; printf
mov  eax, DWORD PTR [esi] ; operator++: get ->next pointer
add  esp, 64
push DWORD PTR [eax+12]
push DWORD PTR [eax+8]
push OFFSET $SG40852 ; 'element at .end(): %d %d'
call edi ; printf
push OFFSET $SG40853 ; '* let''s count from the end:'
call edi ; printf
push DWORD PTR [ebx+12] ; use x and y fields from the node "l" variable
points to
```

```
    push DWORD PTR [ebx+8]
    push OFFSET $SG40860 ; 'element at .end(): %d %d'
    call edi ; printf
    mov  esi, DWORD PTR [ebx+4] ; operator--: get ->prev pointer
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40862 ; '3rd element: %d %d'
    call edi ; printf
    mov  esi, DWORD PTR [esi+4] ; operator--: get ->prev pointer
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40864 ; '2nd element: %d %d'
    call edi ; printf
    mov  eax, DWORD PTR [esi+4] ; operator--: get ->prev pointer
    push DWORD PTR [eax+12]
    push DWORD PTR [eax+8]
    push OFFSET $SG40866 ; '1st element: %d %d'
    call edi ; printf
    add  esp, 64
    push OFFSET $SG40867 ; 'removing last element...'
    call edi ; printf
    mov  edx, DWORD PTR [ebx+4]
    add  esp, 4

    ; prev=next?
    ; it is the only element, garbage one?
    ; if yes, do not delete it!
    cmp  edx, ebx
    je   SHORT $LN349@main
    mov  ecx, DWORD PTR [edx+4]
    mov  eax, DWORD PTR [edx]
    mov  DWORD PTR [ecx], eax
    mov  ecx, DWORD PTR [edx]
    mov  eax, DWORD PTR [edx+4]
    push edx
    mov  DWORD PTR [ecx+4], eax
    call ??3@YAXPAX@Z ; operator delete
    add  esp, 4
    mov  DWORD PTR _l$[esp+32], 2
$LN349@main:
    lea  eax, DWORD PTR _l$[esp+28]
    push eax
    call ?dump_List_val@@YAXPAI@Z ; dump_List_val
    mov  eax, DWORD PTR [ebx]
    add  esp, 4
    mov  DWORD PTR [ebx], ebx
    mov  DWORD PTR [ebx+4], ebx
    cmp  eax, ebx
    je   SHORT $LN412@main
$LL414@main:
    mov  esi, DWORD PTR [eax]
    push eax
    call ??3@YAXPAX@Z ; operator delete
```

```
    add  esp, 4
    mov  eax, esi
    cmp  esi, ebx
    jne  SHORT $LL414@main
$LN412@main:
    push  ebx
    call  ??3@YAXPAX@Z ; operator delete
    add   esp, 4
    xor   eax, eax
    pop   edi
    pop   esi
    pop   ebx
    add   esp, 16
    ret   0
_main ENDP
```

Contrairement à GCC, le code de MSVC alloue l'élément fictif au début de la fonction avec l'aide de la fonction «Buynode», qui est aussi utilisée pour allouer le reste des nœuds (le code de GCC alloue le premier élément dans la pile locale).

Listing 3.115 : La sortie complète

```
* empty list:
_Myhead=0x003CC258, _Mysize=0
ptr=0x003CC258 _Next=0x003CC258 _Prev=0x003CC258 x=6226002 y=4522072
* 3-elements list:
_Myhead=0x003CC258, _Mysize=3
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC2A0 _Prev=0x003CC288 x=1 y=2
ptr=0x003CC2A0 _Next=0x003CC258 _Prev=0x003CC270 x=5 y=6
node at .begin:
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
node at .end:
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072
* let's count from the beginning:
1st element: 3 4
2nd element: 1 2
3rd element: 5 6
element at .end(): 6226002 4522072
* let's count from the end:
element at .end(): 6226002 4522072
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
_Myhead=0x003CC258, _Mysize=2
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC270 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC258 _Prev=0x003CC288 x=1 y=2
```

C++11 std::forward_list

La même chose que std::list, mais simplement chaîné, i.e., ayant seulement le champ «next» dans chaque nœud.

Il a une empreinte mémoire plus faible, mais dons n'offre pas la possibilité de traverser la liste en arrière.

std::vector

Nous appelons `std::vector` un wrapper sûr sur le tableau C PODT[32]. En interne il ressemble à `std::string` (3.21.4 on page 722) : il a un pointeur sur le buffer alloué, un pointeur sur la fin du tableau, et un pointeur sur la fin du buffer alloué.

Les éléments du tableau résident en mémoire adjacents les un aux autres, tout comme un tableau normal (1.26 on page 341). En C++11 il y a une nouvelle méthode appelée `.data()`, qui renvoie un pointeur sur le buffer, comme `.c_str()` dans `std::string`.

Le buffer alloué dans le tas peut être plus large que le tableau lui-même.

Les implémentations de MSVC et GCC sont similaires, seul sont légèrement différents le nom des champs de la structure[33], donc il y a un seul code source qui fonctionne pour les deux compilateurs. Voici encore le code pseudo-C pour afficher la structure de `std::vector` :

```
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <functional>

struct vector_of_ints
{
    // noms MSVC:
    int *Myfirst;
    int *Mylast;
    int *Myend;

    // La structure GCC est la même, mais les noms sont:
    _M_start, _M_finish, _M_end_of_storage
};

void dump(struct vector_of_ints *in)
{
    printf ("_Myfirst=%p, _Mylast=%p, _Myend=%p\n", in->Myfirst, in->Mylast↙
    ↳ , in->Myend);
    size_t size=(in->Mylast-in->Myfirst);
    size_t capacity=(in->Myend-in->Myfirst);
    printf ("size=%d, capacity=%d\n", size, capacity);
    for (size_t i=0; i<size; i++)
        printf ("element %d: %d\n", i, in->Myfirst[i]);
};
```

[32] (C++) Plain Old Data Type
[33] GCC internals: http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01371.html

```
int main()
{
    std::vector<int> c;
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(1);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(2);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(3);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(4);
    dump ((struct vector_of_ints*)(void*)&c);
    c.reserve (6);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(5);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(6);
    dump ((struct vector_of_ints*)(void*)&c);
    printf ("%d\n", c.at(5)); // avec vérifications de limites
    printf ("%d\n", c[8]); // operator[], sans vérifications de limites
};
```

Voici la sortie de ce programme lorsqu'il est compilé dans MSVC:

```
_Myfirst=00000000, _Mylast=00000000, _Myend=00000000
size=0, capacity=0
_Myfirst=0051CF48, _Mylast=0051CF4C, _Myend=0051CF4C
size=1, capacity=1
element 0: 1
_Myfirst=0051CF58, _Mylast=0051CF60, _Myend=0051CF60
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0051C278, _Mylast=0051C284, _Myend=0051C284
size=3, capacity=3
element 0: 1
element 1: 2
element 2: 3
_Myfirst=0051C290, _Mylast=0051C2A0, _Myend=0051C2A0
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0051B180, _Mylast=0051B190, _Myend=0051B198
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0051B180, _Mylast=0051B194, _Myend=0051B198
size=5, capacity=6
element 0: 1
element 1: 2
```

```
element 2: 3
element 3: 4
element 4: 5
_Myfirst=0051B180, _Mylast=0051B198, _Myend=0051B198
size=6, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
6619158
```

On voit qu'il n'y a pas de buffer alloué lorsque `main()` débute. Après le premier appel à `push_back()`, un buffer est alloué. Et puis, après chaque appel à `push_back()`, la taille du tableau et la taille du buffer (*capacity*) sont augmentées. Mais l'adresse du buffer change aussi, car `push_back()` ré-alloue le buffer dans le tas à chaque fois. C'est une opération coûteuse, c'est pourquoi il est très important de prévoir la taille du tableau dans le futur et de lui réserver assez d'espace avec la méthode `.reserve()`.

Le dernier nombre est du déchet: il n'y a pas d'élément du tableau à cet endroit, donc un nombre aléatoire est affiché. Ceci illustre le fait que `operator[]` de `std::vector` ne vérifie pas si l'index est dans le limites du tableau. La méthode plus lente `.at()`, toutefois, fait cette vérification et envoie une exception `std::out_of_range` en cas d'erreur.

Regardons le code:

Listing 3.116 : MSVC 2012 /GS- /Ob1

```
$SG52650 DB '%d', 0aH, 00H
$SG52651 DB '%d', 0aH, 00H

_this$ = -4 ; size = 4
__Pos$ = 8  ; size = 4
?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z PROC ;
    std::vector<int,std::allocator<int> >::at, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR _this$[ebp]
    mov  edx, DWORD PTR [eax+4]
    sub  edx, DWORD PTR [ecx]
    sar  edx, 2
    cmp  edx, DWORD PTR __Pos$[ebp]
    ja   SHORT $LN1@at
    push OFFSET ??_C@_0BM@NMJKDPPO@invalid?5vector?$DMT?$DO?5subscript?$AA@
    call DWORD PTR __imp_?_Xout_of_range@std@@YAXPBD@Z
$LN1@at:
```

```
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR [eax]
    mov  edx, DWORD PTR __Pos$[ebp]
    lea  eax, DWORD PTR [ecx+edx*4]
$LN3@at:
    mov  esp, ebp
    pop  ebp
    ret  4
?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ENDP ;
    std::vector<int,std::allocator<int> >::at

_c$ = -36 ; size = 12
$T1 = -24 ; size = 4
$T2 = -20 ; size = 4
$T3 = -16 ; size = 4
$T4 = -12 ; size = 4
$T5 = -8  ; size = 4
$T6 = -4  ; size = 4
_main PROC
    push ebp
    mov  ebp, esp
    sub  esp, 36
    mov  DWORD PTR _c$[ebp], 0     ; Myfirst
    mov  DWORD PTR _c$[ebp+4], 0   ; Mylast
    mov  DWORD PTR _c$[ebp+8], 0   ; Myend
    lea  eax, DWORD PTR _c$[ebp]
    push eax
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T6[ebp], 1
    lea  ecx, DWORD PTR $T6[ebp]
    push ecx
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ;
    std::vector<int,std::allocator<int> >::push_back
    lea  edx, DWORD PTR _c$[ebp]
    push edx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T5[ebp], 2
    lea  eax, DWORD PTR $T5[ebp]
    push eax
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ;
    std::vector<int,std::allocator<int> >::push_back
    lea  ecx, DWORD PTR _c$[ebp]
    push ecx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T4[ebp], 3
    lea  edx, DWORD PTR $T4[ebp]
    push edx
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ;
```

```
std::vector<int,std::allocator<int> >::push_back
    lea  eax, DWORD PTR _c$[ebp]
    push eax
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T3[ebp], 4
    lea  ecx, DWORD PTR $T3[ebp]
    push ecx
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ;
std::vector<int,std::allocator<int> >::push_back
    lea  edx, DWORD PTR _c$[ebp]
    push edx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    push 6
    lea  ecx, DWORD PTR _c$[ebp]
    call ?reserve@?$vector@HV?$allocator@H@std@@@std@@QAEXI@Z ;
std::vector<int,std::allocator<int> >::reserve
    lea  eax, DWORD PTR _c$[ebp]
    push eax
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T2[ebp], 5
    lea  ecx, DWORD PTR $T2[ebp]
    push ecx
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ;
std::vector<int,std::allocator<int> >::push_back
    lea  edx, DWORD PTR _c$[ebp]
    push edx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T1[ebp], 6
    lea  eax, DWORD PTR $T1[ebp]
    push eax
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ;
std::vector<int,std::allocator<int> >::push_back
    lea  ecx, DWORD PTR _c$[ebp]
    push ecx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    push 5
    lea  ecx, DWORD PTR _c$[ebp]
    call ?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ;
std::vector<int,std::allocator<int> >::at
    mov  edx, DWORD PTR [eax]
    push edx
    push OFFSET $SG52650 ; '%d'
    call DWORD PTR __imp__printf
    add  esp, 8
    mov  eax, 8
    shl  eax, 2
    mov  ecx, DWORD PTR _c$[ebp]
```

```
    mov  edx, DWORD PTR [ecx+eax]
    push edx
    push OFFSET $SG52651 ; '%d'
    call DWORD PTR __imp__printf
    add  esp, 8
    lea  ecx, DWORD PTR _c$[ebp]
    call ?_Tidy@?$vector@HV?$allocator@H@std@@@std@@IAEXXZ ;
    std::vector<int,std::allocator<int> >::_Tidy
    xor  eax, eax
    mov  esp, ebp
    pop  ebp
    ret  0
_main ENDP
```

Nous voyons que la méthode `.at()` vérifie les limites et envoie une exception en cas d'erreur. Le nombre que le dernier appel à `printf()` affiche est pris de la mémoire, sans aucune vérification.

On peut se demander pourquoi ne pas utiliser des variables comme «size» et «capacity», comme c'est fait dans `std::string`. Probablement que c'est fait comme cela pour avoir une vérification des limites plus rapide.

Le code que GCC génère est en général presque le même, mais la méthode `.at()` est mise en ligne:

Listing 3.117 : GCC 4.8.1 -fno-inline-small-functions -O1

```
main proc near
     push ebp
     mov  ebp, esp
     push edi
     push esi
     push ebx
     and  esp, 0FFFFFFF0h
     sub  esp, 20h
     mov  dword ptr [esp+14h], 0
     mov  dword ptr [esp+18h], 0
     mov  dword ptr [esp+1Ch], 0
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  dword ptr [esp+10h], 1
     lea  eax, [esp+10h]
     mov  [esp+4], eax
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _ZNSt6vectorIiSaIiEE9push_backERKi ;
    std::vector<int,std::allocator<int>>::push_back(int  const&)
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  dword ptr [esp+10h], 2
     lea  eax, [esp+10h]
     mov  [esp+4], eax
     lea  eax, [esp+14h]
```

```
     mov  [esp], eax
     call _ZNSt6vectorIiSaIiEE9push_backERKi ;
    std::vector<int,std::allocator<int>>::push_back(int  const&)
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  dword ptr [esp+10h], 3
     lea  eax, [esp+10h]
     mov  [esp+4], eax
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _ZNSt6vectorIiSaIiEE9push_backERKi ;
    std::vector<int,std::allocator<int>>::push_back(int  const&)
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  dword ptr [esp+10h], 4
     lea  eax, [esp+10h]
     mov  [esp+4], eax
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _ZNSt6vectorIiSaIiEE9push_backERKi ;
    std::vector<int,std::allocator<int>>::push_back(int  const&)
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  ebx, [esp+14h]
     mov  eax, [esp+1Ch]
     sub  eax, ebx
     cmp  eax, 17h
     ja   short loc_80001CF
     mov  edi, [esp+18h]
     sub  edi, ebx
     sar  edi, 2
     mov  dword ptr [esp], 18h
     call _Znwj           ; operator new(uint)
     mov  esi, eax
     test edi, edi
     jz   short loc_80001AD
     lea  eax, ds:0[edi*4]
     mov  [esp+8], eax    ; n
     mov  [esp+4], ebx    ; src
     mov  [esp], esi      ; dest
     call memmove

loc_80001AD: ; CODE XREF: main+F8
     mov  eax, [esp+14h]
     test eax, eax
     jz   short loc_80001BD
     mov  [esp], eax      ; void *
     call _ZdlPv          ; operator delete(void *)

loc_80001BD: ; CODE XREF: main+117
     mov  [esp+14h], esi
```

```
    lea  eax, [esi+edi*4]
    mov  [esp+18h], eax
    add  esi, 18h
    mov  [esp+1Ch], esi

loc_80001CF: ; CODE XREF: main+DD
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov  dword ptr [esp+10h], 5
    lea  eax, [esp+10h]
    mov  [esp+4], eax
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi ;
   std::vector<int,std::allocator<int>>::push_back(int  const&)
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov  dword ptr [esp+10h], 6
    lea  eax, [esp+10h]
    mov  [esp+4], eax
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi ;
   std::vector<int,std::allocator<int>>::push_back(int  const&)
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov  eax, [esp+14h]
    mov  edx, [esp+18h]
    sub  edx, eax
    cmp  edx, 17h
    ja   short loc_8000246
    mov  dword ptr [esp], offset aVector_m_range ; "vector::_M_range_check"
    call _ZSt20__throw_out_of_rangePKc ;
   std::__throw_out_of_range(char  const*)

loc_8000246:                            ; CODE XREF: main+19C
    mov  eax, [eax+14h]
    mov  [esp+8], eax
    mov  dword ptr [esp+4], offset aD ; "%d\n"
    mov  dword ptr [esp], 1
    call __printf_chk
    mov  eax, [esp+14h]
    mov  eax, [eax+20h]
    mov  [esp+8], eax
    mov  dword ptr [esp+4], offset aD ; "%d\n"
    mov  dword ptr [esp], 1
    call __printf_chk
    mov  eax, [esp+14h]
    test eax, eax
    jz   short loc_80002AC
    mov  [esp], eax     ; void *
```

```
     call _ZdlPv          ; operator delete(void *)
     jmp  short loc_80002AC

     mov  ebx, eax
     mov  edx, [esp+14h]
     test edx, edx
     jz   short loc_80002A4
     mov  [esp], edx      ; void *
     call _ZdlPv          ; operator delete(void *)

loc_80002A4: ; CODE XREF: main+1FE
     mov  [esp], ebx
     call _Unwind_Resume


loc_80002AC: ; CODE XREF: main+1EA
             ; main+1F4
     mov  eax, 0
     lea  esp, [ebp-0Ch]
     pop  ebx
     pop  esi
     pop  edi
     pop  ebp

locret_80002B8: ; DATA XREF: .eh_frame:08000510
                ; .eh_frame:080005BC
     retn
main endp
```

.reserve() est aussi mise en ligne. Elle appelle new() si le buffer est trop petit pour la nouvelle taille, appelle memmove() pour copier le contenu du buffer et appelle delete() pour libérer l'ancien buffer.

Regardons aussi ce que le programme affiche s'il est compilé avec GCC:

```
_Myfirst=0x(nil), _Mylast=0x(nil), _Myend=0x(nil)
size=0, capacity=0
_Myfirst=0x8257008, _Mylast=0x825700c, _Myend=0x825700c
size=1, capacity=1
element 0: 1
_Myfirst=0x8257018, _Mylast=0x8257020, _Myend=0x8257020
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0x8257028, _Mylast=0x8257034, _Myend=0x8257038
size=3, capacity=4
element 0: 1
element 1: 2
element 2: 3
_Myfirst=0x8257028, _Mylast=0x8257038, _Myend=0x8257038
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3
```

```
element 3: 4
_Myfirst=0x8257040, _Mylast=0x8257050, _Myend=0x8257058
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0x8257040, _Mylast=0x8257054, _Myend=0x8257058
size=5, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
_Myfirst=0x8257040, _Mylast=0x8257058, _Myend=0x8257058
size=6, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
0
```

Nous repérons que la taille du buffer grossit d'une manière différente qu'avec MSVC.

Une simple expérimentation montre que l'implémentation de MSVC augmente le buffer de ~50% à chaque fois qu'il a besoin d'être augmenté, tandis que le code de GCC l'augmente de 100% à chaque fois, i.e., le double.

std::map and std::set

L'arbre binaire est une autre structure de données fondamentale.

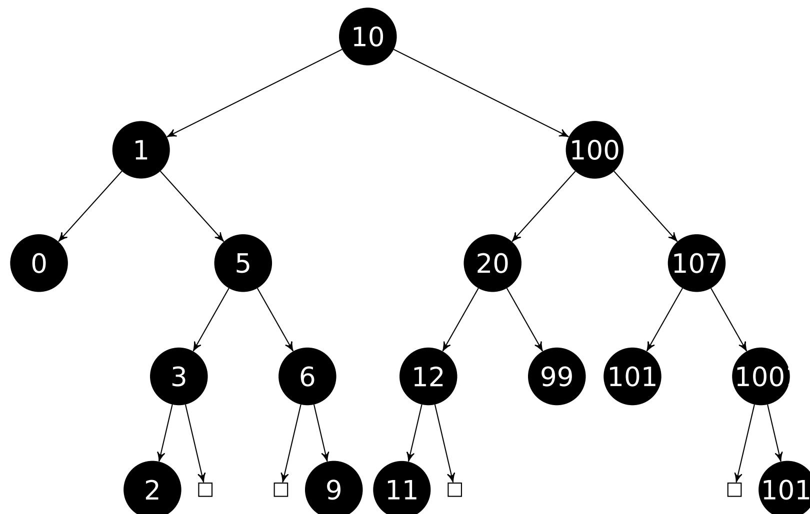
Comme son nom l'indique, c'est un arbre où chaque nœud a au plus 2 liens sur un autre nœud. Chaque nœud a une clef et/ou un valeur: `std::set` fourni seulement une clef dans chaque nœud, `std::map` fourni à la fois une clef et une valeur dans chaque nœud

Les arbres binaire sont générallement la structure utilisée dans l'implémentation des dictionnaires de clef-valeurs ((AKA «tableaux associatifs»).

Il y a au moins ces trois propriétés importante qu'un arbre binaire possède:

- Toutes les clefs sont toujours stockées sous une forme triée.
- Les clefs de tout type peuvent être stockées facilement. Les algorithmes de l'arbre binaire ne sont pas conscients du type de clef, seule une fonction de comparaison de clef est nécessaire.
- Trouver une clef particulière est relativement rapide comparé aux listes chaînées et tableaux.

Voici un exemple très simple: stockons ces nombres dans un arbre binaire: 0, 1, 2, 3, 5, 6, 9, 10, 11, 12, 20, 99, 100, 101, 107, 1001, 1010.



Toutes les clefs qui sont plus petites que la valeur de la clef du nœud sont stockées du côté gauche.

Toutes les clefs qui sont plus grandes que la valeur de la clef du nœud sont stockées du côté droit.

Ainsi, l'algorithme de recherche est simple: si la valeur que vous cherchez est plus petite que la valeur de la clef du nœud courant: déplacez à gauche, si elle plus grande: déplacez à droite, stoppez si la valeur cherchée est égale à la valeur de la clef du nœud.

C'est pourquoi l'algorithme de recherche peut chercher des nombres, des chaînes de texte, etc., tant que la fonction de comparaison de clef est fourni.

Toutes les clefs ont des valeurs uniques.

En ayant cela, il faut ≈ $\log_2 n$ itérations pour trouver une clef dans un arbre binaire équilibré avec n clef. Ceci implique que ≈ 10 itérations sont pour ≈ 1000 keys, ou ≈ 13 itérations pour ≈ 10000 clefs.

Pas mal, mais l'arbre doit toujours être équilibré pour cela: i.e., les clefs doivent être distribuées uniformément à chaque niveaux. Les opérations d'insertion et de suppression font un peut de maintenance pour garder l'arbre dans un état équilibré.

Il y a plusieurs algorithme d'équilibrage disponible, incluant l'arbre AVL et l'arbre red-black.

La dernière étend chaque nœud avec une valeur de «couleur» opur simplifier le processus de rééquilibrage, de ce fait, chaque nœud peut être rouge ou noir.

À la fois les implémentations des templates `std::map` et `std::set` de GCC et MSVC utilisent les arbres red-black.

`std::set` a seulement des clefs. `std::map` est la version étendue de `std::set` : il a aussi une valeur à chaque nœud.

MSVC

```
#include <map>
#include <set>
#include <string>
#include <iostream>

// La structure n'est pas paquée! Chaque champ occupe 4 octets.
struct tree_node
{
    struct tree_node *Left;
    struct tree_node *Parent;
    struct tree_node *Right;
    char Color; // 0 - Red, 1 - Black
    char Isnil;
    //std::pair Myval;
    unsigned int first; // appelé Myval dans std::set
    const char *second; // non présent dans std::set
};

struct tree_struct
{
    struct tree_node *Myhead;
    size_t Mysize;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool traverse)
{
    printf ("ptr=0x%p Left=0x%p Parent=0x%p Right=0x%p Color=%d Isnil=%d\n"↙
    ↪ ,
            n, n->Left, n->Parent, n->Right, n->Color, n->Isnil);
    if (n->Isnil==0)
    {
        if (is_set)
            printf ("first=%d\n", n->first);
        else
            printf ("first=%d second=[%s]\n", n->first, n->second);
    }

    if (traverse)
    {
        if (n->Isnil==1)
            dump_tree_node (n->Parent, is_set, true);
        else
        {
            if (n->Left->Isnil==0)
                dump_tree_node (n->Left, is_set, true);
            if (n->Right->Isnil==0)
                dump_tree_node (n->Right, is_set, true);
        };
    };
};
```

```
const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    if (is_set)
        printf ("%d\n", n->first);
    else
        printf ("%d [%s]\n", n->first, n->second);
    if (n->Left->Isnil==0)
    {
        printf ("%.*sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Left, is_set);
    };
    if (n->Right->Isnil==0)
    {
        printf ("%.*sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Right, is_set);
    };
};

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, Myhead=0x%p, Mysize=%d\n", m, m->Myhead, m->Mysize);
    dump_tree_node (m->Myhead, is_set, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->Myhead->Parent, is_set);
};

int main()
{
    // map

    std::map<int, const char*> m;

    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    m[107]="one hundred seven";
    m[0]="zero";
    m[1]="one";
    m[6]="six";
    m[99]="ninety-nine";
    m[5]="five";
    m[11]="eleven";
    m[1001]="one thousand one";
    m[1010]="one thousand ten";
    m[2]="two";
    m[9]="nine";
    printf ("dumping m as map:\n");
```

```
    dump_map_and_set ((struct tree_struct *)(void*)&m, false);

    std::map<int, const char*>::iterator it1=m.begin();
    printf ("m.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, false);
    it1=m.end();
    printf ("m.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, false);

    // set

    std::set<int> s;
    s.insert(123);
    s.insert(456);
    s.insert(11);
    s.insert(12);
    s.insert(100);
    s.insert(1001);
    printf ("dumping s as set:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s, true);
    std::set<int>::iterator it2=s.begin();
    printf ("s.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, false);
    it2=s.end();
    printf ("s.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, false);
};
```

Listing 3.118 : MSVC 2012

```
dumping m as map:
ptr=0x0020FE04, Myhead=0x005BB3A0, Mysize=17
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB580 Color=1 ↙
    ↳ Isnil=1
ptr=0x005BB3C0 Left=0x005BB4C0 Parent=0x005BB3A0 Right=0x005BB440 Color=1 ↙
    ↳ Isnil=0
first=10 second=[ten]
ptr=0x005BB4C0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB520 Color=1 ↙
    ↳ Isnil=0
first=1 second=[one]
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0x005BB3A0 Color=1 ↙
    ↳ Isnil=0
first=0 second=[zero]
ptr=0x005BB520 Left=0x005BB400 Parent=0x005BB4C0 Right=0x005BB4E0 Color=0 ↙
    ↳ Isnil=0
first=5 second=[five]
ptr=0x005BB400 Left=0x005BB5A0 Parent=0x005BB520 Right=0x005BB3A0 Color=1 ↙
    ↳ Isnil=0
first=3 second=[three]
ptr=0x005BB5A0 Left=0x005BB3A0 Parent=0x005BB400 Right=0x005BB3A0 Color=0 ↙
    ↳ Isnil=0
first=2 second=[two]
ptr=0x005BB4E0 Left=0x005BB3A0 Parent=0x005BB520 Right=0x005BB5C0 Color=1 ↙
    ↳ Isnil=0
```

```
first=6 second=[six]
ptr=0x005BB5C0 Left=0x005BB3A0 Parent=0x005BB4E0 Right=0x005BB3A0 Color=0 ↙
    ↳ Isnil=0
first=9 second=[nine]
ptr=0x005BB440 Left=0x005BB3E0 Parent=0x005BB3C0 Right=0x005BB480 Color=1 ↙
    ↳ Isnil=0
first=100 second=[one hundred]
ptr=0x005BB3E0 Left=0x005BB460 Parent=0x005BB440 Right=0x005BB500 Color=0 ↙
    ↳ Isnil=0
first=20 second=[twenty]
ptr=0x005BB460 Left=0x005BB540 Parent=0x005BB3E0 Right=0x005BB3A0 Color=1 ↙
    ↳ Isnil=0
first=12 second=[twelve]
ptr=0x005BB540 Left=0x005BB3A0 Parent=0x005BB460 Right=0x005BB3A0 Color=0 ↙
    ↳ Isnil=0
first=11 second=[eleven]
ptr=0x005BB500 Left=0x005BB3A0 Parent=0x005BB3E0 Right=0x005BB3A0 Color=1 ↙
    ↳ Isnil=0
first=99 second=[ninety-nine]
ptr=0x005BB480 Left=0x005BB420 Parent=0x005BB440 Right=0x005BB560 Color=0 ↙
    ↳ Isnil=0
first=107 second=[one hundred seven]
ptr=0x005BB420 Left=0x005BB3A0 Parent=0x005BB480 Right=0x005BB3A0 Color=1 ↙
    ↳ Isnil=0
first=101 second=[one hundred one]
ptr=0x005BB560 Left=0x005BB3A0 Parent=0x005BB480 Right=0x005BB580 Color=1 ↙
    ↳ Isnil=0
first=1001 second=[one thousand one]
ptr=0x005BB580 Left=0x005BB3A0 Parent=0x005BB560 Right=0x005BB3A0 Color=0 ↙
    ↳ Isnil=0
first=1010 second=[one thousand ten]
As a tree:
root----10 [ten]
        L-------1 [one]
                L-------0 [zero]
                R-------5 [five]
                        L-------3 [three]
                                L-------2 [two]
                        R-------6 [six]
                                R-------9 [nine]
        R-------100 [one hundred]
                L-------20 [twenty]
                        L-------12 [twelve]
                                L-------11 [eleven]
                        R-------99 [ninety-nine]
                R-------107 [one hundred seven]
                        L-------101 [one hundred one]
                        R-------1001 [one thousand one]
                                R-------1010 [one thousand ten]
m.begin():
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0x005BB3A0 Color=1 ↙
    ↳ Isnil=0
first=0 second=[zero]
```

```
m.end():
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB580 Color=1 ↙
    ↳ Isnil=1

dumping s as set:
ptr=0x0020FDFC, Myhead=0x005BB5E0, Mysize=6
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB6A0 Color=1 ↙
    ↳ Isnil=1
ptr=0x005BB600 Left=0x005BB660 Parent=0x005BB5E0 Right=0x005BB620 Color=1 ↙
    ↳ Isnil=0
first=123
ptr=0x005BB660 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB680 Color=1 ↙
    ↳ Isnil=0
first=12
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 ↙
    ↳ Isnil=0
first=11
ptr=0x005BB680 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 ↙
    ↳ Isnil=0
first=100
ptr=0x005BB620 Left=0x005BB5E0 Parent=0x005BB600 Right=0x005BB6A0 Color=1 ↙
    ↳ Isnil=0
first=456
ptr=0x005BB6A0 Left=0x005BB5E0 Parent=0x005BB620 Right=0x005BB5E0 Color=0 ↙
    ↳ Isnil=0
first=1001
As a tree:
root----123
        L-------12
                L-------11
                R-------100
        R-------456
                R-------1001
s.begin():
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 ↙
    ↳ Isnil=0
first=11
s.end():
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB6A0 Color=1 ↙
    ↳ Isnil=1
```

La structure n'est pas paquée, donc chaque valeur *char* occupe 4 octets.

`std::set`

[Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, *Introduction to Algorithms, Third Edition*, (2009)].

[34].

GCC

[34] http://www.ethoberon.ethz.ch/WirthPubl/AD.pdf

```
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair
{
    int key;
    const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool traverse, bool ↙
    ↳ dump_keys_and_values)
{
    printf ("ptr=0x%p M_left=0x%p M_parent=0x%p M_right=0x%p M_color=%d\n",
            n, n->M_left, n->M_parent, n->M_right, n->M_color);

    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    if (dump_keys_and_values)
    {
        if (is_set)
            printf ("key=%d\n", *(int*)point_after_struct);
        else
        {
            struct map_pair *p=(struct map_pair *)point_after_struct;
            printf ("key=%d value=[%s]\n", p->key, p->value);
        };
    };

    if (traverse==false)
        return;

    if (n->M_left)
        dump_tree_node (n->M_left, is_set, traverse, dump_keys_and_values);
    if (n->M_right)
        dump_tree_node (n->M_right, is_set, traverse, dump_keys_and_values)↙
```

```
    ↳ ;
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    if (is_set)
        printf ("%d\n", *(int*)point_after_struct);
    else
    {
        struct map_pair *p=(struct map_pair *)point_after_struct;
        printf ("%d [%s]\n", p->key, p->value);
    }

    if (n->M_left)
    {
        printf ("%.*sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left, is_set);
    };
    if (n->M_right)
    {
        printf ("%.*sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right, is_set);
    };
};

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, M_key_compare=0x%x, M_header=0x%p, M_node_count=%d\n↙
    ↳ ",
        m, m->M_key_compare, &m->M_header, m->M_node_count);
    dump_tree_node (m->M_header.M_parent, is_set, true, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent, is_set);
};

int main()
{
    // map

    std::map<int, const char*> m;

    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    m[107]="one hundred seven";
```

```
    m[0]="zero";
    m[1]="one";
    m[6]="six";
    m[99]="ninety-nine";
    m[5]="five";
    m[11]="eleven";
    m[1001]="one thousand one";
    m[1010]="one thousand ten";
    m[2]="two";
    m[9]="nine";

    printf ("dumping m as map:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&m, false);

    std::map<int, const char*>::iterator it1=m.begin();
    printf ("m.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, false, true);
    it1=m.end();
    printf ("m.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, false, false)↙
    ↳ ;

    // set

    std::set<int> s;
    s.insert(123);
    s.insert(456);
    s.insert(11);
    s.insert(12);
    s.insert(100);
    s.insert(1001);
    printf ("dumping s as set:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s, true);
    std::set<int>::iterator it2=s.begin();
    printf ("s.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, false, true);
    it2=s.end();
    printf ("s.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, false, false);
};
```

Listing 3.119 : GCC 4.8.1

```
dumping m as map:
ptr=0x0028FE3C, M_key_compare=0x402b70, M_header=0x0028FE40, M_node_count↙
    ↳ =17
ptr=0x007A4988 M_left=0x007A4C00 M_parent=0x0028FE40 M_right=0x007A4B80 ↙
    ↳ M_color=1
key=10 value=[ten]
ptr=0x007A4C00 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0x007A4C60 ↙
    ↳ M_color=1
key=1 value=[one]
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0x00000000 ↙
    ↳ M_color=1
```

```
key=0 value=[zero]
ptr=0x007A4C60 M_left=0x007A4B40 M_parent=0x007A4C00 M_right=0x007A4C20 ↙
    ↳ M_color=0
key=5 value=[five]
ptr=0x007A4B40 M_left=0x007A4CE0 M_parent=0x007A4C60 M_right=0x00000000 ↙
    ↳ M_color=1
key=3 value=[three]
ptr=0x007A4CE0 M_left=0x00000000 M_parent=0x007A4B40 M_right=0x00000000 ↙
    ↳ M_color=0
key=2 value=[two]
ptr=0x007A4C20 M_left=0x00000000 M_parent=0x007A4C60 M_right=0x007A4D00 ↙
    ↳ M_color=1
key=6 value=[six]
ptr=0x007A4D00 M_left=0x00000000 M_parent=0x007A4C20 M_right=0x00000000 ↙
    ↳ M_color=0
key=9 value=[nine]
ptr=0x007A4B80 M_left=0x007A49A8 M_parent=0x007A4988 M_right=0x007A4BC0 ↙
    ↳ M_color=1
key=100 value=[one hundred]
ptr=0x007A49A8 M_left=0x007A4BA0 M_parent=0x007A4B80 M_right=0x007A4C40 ↙
    ↳ M_color=0
key=20 value=[twenty]
ptr=0x007A4BA0 M_left=0x007A4C80 M_parent=0x007A49A8 M_right=0x00000000 ↙
    ↳ M_color=1
key=12 value=[twelve]
ptr=0x007A4C80 M_left=0x00000000 M_parent=0x007A4BA0 M_right=0x00000000 ↙
    ↳ M_color=0
key=11 value=[eleven]
ptr=0x007A4C40 M_left=0x00000000 M_parent=0x007A49A8 M_right=0x00000000 ↙
    ↳ M_color=1
key=99 value=[ninety-nine]
ptr=0x007A4BC0 M_left=0x007A4B60 M_parent=0x007A4B80 M_right=0x007A4CA0 ↙
    ↳ M_color=0
key=107 value=[one hundred seven]
ptr=0x007A4B60 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0x00000000 ↙
    ↳ M_color=1
key=101 value=[one hundred one]
ptr=0x007A4CA0 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0x007A4CC0 ↙
    ↳ M_color=1
key=1001 value=[one thousand one]
ptr=0x007A4CC0 M_left=0x00000000 M_parent=0x007A4CA0 M_right=0x00000000 ↙
    ↳ M_color=0
key=1010 value=[one thousand ten]
As a tree:
root----10 [ten]
        L-------1 [one]
                L-------0 [zero]
                R-------5 [five]
                        L-------3 [three]
                                L-------2 [two]
                        R-------6 [six]
                                R-------9 [nine]
        R-------100 [one hundred]
```

```
                L-------20 [twenty]
                        L-------12 [twelve]
                                L-------11 [eleven]
                        R-------99 [ninety-nine]
                R-------107 [one hundred seven]
                        L-------101 [one hundred one]
                        R-------1001 [one thousand one]
                                R-------1010 [one thousand ten]
m.begin():
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0x00000000 ↙
    ↳ M_color=1
key=0 value=[zero]
m.end():
ptr=0x0028FE40 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0x007A4CC0 ↙
    ↳ M_color=0

dumping s as set:
ptr=0x0028FE20, M_key_compare=0x8, M_header=0x0028FE24, M_node_count=6
ptr=0x007A1E80 M_left=0x01D5D890 M_parent=0x0028FE24 M_right=0x01D5D850 ↙
    ↳ M_color=1
key=123
ptr=0x01D5D890 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0x01D5D8B0 ↙
    ↳ M_color=1
key=12
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 ↙
    ↳ M_color=0
key=11
ptr=0x01D5D8B0 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 ↙
    ↳ M_color=0
key=100
ptr=0x01D5D850 M_left=0x00000000 M_parent=0x007A1E80 M_right=0x01D5D8D0 ↙
    ↳ M_color=1
key=456
ptr=0x01D5D8D0 M_left=0x00000000 M_parent=0x01D5D850 M_right=0x00000000 ↙
    ↳ M_color=0
key=1001
As a tree:
root----123
        L-------12
                L-------11
                R-------100
        R-------456
                R-------1001
s.begin():
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 ↙
    ↳ M_color=0
key=11
s.end():
ptr=0x0028FE24 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0x01D5D8D0 ↙
    ↳ M_color=0
```

35.

Démo de rééquilibrage (GCC)

Il y a aussi une démo nous montrant comment un arbre est rééquilibré après quelques insertions.

Listing 3.120 : GCC

```
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair
{
    int key;
    const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    printf ("%d\n", *(int*)point_after_struct);

    if (n->M_left)
    {
        printf ("%.*sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left);
    };
    if (n->M_right)
```

[35] http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/stl__tree_8h-source.html

```
    {
        printf ("%.*sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right);
    };
};

void dump_map_and_set(struct tree_struct *m)
{
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent);
};

int main()
{
    std::set<int> s;
    s.insert(123);
    s.insert(456);
    printf ("123, 456 has been inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(11);
    s.insert(12);
    printf ("\n");
    printf ("11, 12 has been inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(100);
    s.insert(1001);
    printf ("\n");
    printf ("100, 1001 has been inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(667);
    s.insert(1);
    s.insert(4);
    s.insert(7);
    printf ("\n");
    printf ("667, 1, 4, 7 has been inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    printf ("\n");
};
```

Listing 3.121 : GCC 4.8.1

```
123, 456 has been inserted
root----123
        R-------456

11, 12 has been inserted
root----123
        L-------11
                R-------12
        R-------456

100, 1001 has been inserted
root----123
        L-------12
```

```
                L-------11
                R-------100
        R-------456
                R-------1001

667, 1, 4, 7 has been inserted
root----12
        L-------4
                L-------1
                R-------11
                        L-------7
        R-------123
                L-------100
                R-------667
                        L-------456
                        R-------1001
```

3.21.5 Mémoire

Vous pouvez parfois entendre de la part de programmeurs C++ «allouer la mémoire sur la pile» et/ou «allouer la mémoire sur le tas».

Allouer un objet *sur la pile* :

```
void f()
{
        ...

        Class o=Class(...);

        ...
};
```

La mémoire de l'objet (ou de la structure) est allouée sur le pile, en utilisant un simple décalage de SP. La mémoire est allouée jusqu'à la sortie de la fonction, ou, plus précisément, à la fin du *scope*—SP est remis à son état (comme au début de la fonction) et le destructeur de *Class* est appelé. De la même manière, la mémoire allouée pour une structure en C est désallouée à la sortie de la fonction.

Allouer un objet *dans le tas* :

```
void f1()
{
        ...

        Class *o=new Class(...);

        ...
};

void f2()
{
        ...
```

```
        delete o;

        ...
};
```

Ceci est la même chose que d'allouer de la mémoire pour une structure en utilisant un appel à *malloc()*. En fait, *new* en C++ est un wrapper pour *malloc()*, et *delete* est un wrapper pour *free()*. Puisque le bloc de mémoire a été allouée sur le tas, il doit être désalloué explicitement, en utilisant *delete*. Le destructeur de classe sera appelé automatiquement juste avant ce moment.

Quelle méthode est la meilleure? L'allocation *isur la pile* est très rapide, et bon pour les petits, à durée de vie courte objets, qui seront utilisés seulement dans la fonction courante.

L'allocation *sur le heap* est plus lente, et meilleure pour des objets à longue durée de vie, qui seront utilisés dans plusieurs fonctions. Aussi, les objets alloués sur le tas sont sujets à la fuite de mémoire, car ils doivent être libérés explicitement, mais on peut oublier de le faire.

De toutes façons, ceci est une affaire de goût.

3.22 Index de tableau négatifs

Il est possible d'accéder à l'espace *avant* un tableau en donnant un index négatif, e.g., $array[-1]$.

3.22.1 Accéder à une chaîne depuis la fin

Python LP permet d'accéder aux tableaux depuis la fin. Par exemple, *string[-1]* renvoie le dernier caractère, *string[-2]* renvoie le pénultième, etc. Difficile à croire, mais ceci est aussi possible en C/C++ :

```
#include <string.h>
#include <stdio.h>

int main()
{
        char *s="Hello, world!";
        char *s_end=s+strlen(s);

        printf ("last character: %c\n", s_end[-1]);
        printf ("penultimate character: %c\n", s_end[-2]);
};
```

Ça fonctionne, mais *s_end* doit toujours avoir l'adresse du zéro en fin de la chaîne *s*. Si la taille de la chaîne *s* est modifiée, *s_end* doit être modifié aussi.

L'astuce est douteuse, mais, encore une fois, ceci est une démonstration d'indices négatifs.

3.22.2 Accéder à un bloc quelconque depuis la fin

Rappelons d'abord pourquoi la pile grossit en arrière (1.9.1 on page 43). Il y a une sorte de bloc en mémoire et vous voulez y stocker à la fois le heap et la pile, sans savoir comment ils vont grossir pendant l'exécution.

Vous pouvez mettre un pointeur de *heap* au début du bloc, puis vous mettez un pointeur de *pile* à la fin du bloc (*heap* + *size_of_block*), et vous pouvez accéder au *n-ième* élément de la pile avec *stack[-n]*. Par exemple, *stack[-1]* pour le 1er élément, *stack[-2]* pour le 2nd, etc.

Ceci fonctionnera de la même façon que notre astuce d'accéder la chaîne depuis la fin.

Vous pouvez facilement vérifier si les structures n'ont pas commencé à se recouvrir: il suffit d'être sûr que l'adresse du dernier élément dans le *heap* est plus bas que le dernier élément de la *pile*.

Malheureusement, l'index -0 ne fonctionnera pas, puisque la représentation des nombres négatifs en complément à deux ne permet pas de zéro négatif. donc il ne peut pas être distingué d'un zéro positif.

Ceci est aussi mentionné dans "Transaction processing", Jim Gray, 1993, chapitre "The Tuple-Oriented File System", p. 755.

3.22.3 Tableaux commençants à 1

Fortran et Mathematica définissent le premier élément d'un tableau avec l'indice 1, sans doute parce que c'est la tradition en mathématiques. D'autres LPs comme C/C++ le définisse avec l'indice 0. Lequel est le meilleur? Edsger W. Dijkstra prétend que le dernier est le meilleur [36].

Mais les programmeurs peuvent toujours avoir l'habitude après le Fortran, donc avec ce petit truc, il est possible d'accéder au premier élément en C/C++ en utilisant l'indice 1:

```
#include <stdio.h>

int main()
{
        int random_value=0x11223344;
        unsigned char array[10];
        int i;
        unsigned char *fakearray=&array[-1];

        for (i=0; i<10; i++)
                array[i]=i;

        printf ("first element %d\n", fakearray[1]);
        printf ("second element %d\n", fakearray[2]);
        printf ("last element %d\n", fakearray[10]);
```

[36] Voir https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html

```
        printf ("array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array⤦
    ↪ [-4]=%02X\n",
                array[-1],
                array[-2],
                array[-3],
                array[-4]);
};
```

Listing 3.122 : MSVC 2010 sans optimisation

```
$SG2751 DB      'first element %d', 0aH, 00H
$SG2752 DB      'second element %d', 0aH, 00H
$SG2753 DB      'last element %d', 0aH, 00H
$SG2754 DB      'array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4'
        DB      ']=%02X', 0aH, 00H

_fakearray$ = -24               ; taille = 4
_random_value$ = -20    ; taille = 4
_array$ = -16           ; taille = 10
_i$ = -4                ; taille = 4
_main   PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 24
        mov     DWORD PTR _random_value$[ebp], 287454020 ; 11223344H
        ; définir fakearray[] un octet avant array[]
        lea     eax, DWORD PTR _array$[ebp]
        add     eax, -1 ; eax=eax-1
        mov     DWORD PTR _fakearray$[ebp], eax
        mov     DWORD PTR _i$[ebp], 0
        jmp     SHORT $LN3@main
        ; remplir array[] avec 0..9
$LN2@main:
        mov     ecx, DWORD PTR _i$[ebp]
        add     ecx, 1
        mov     DWORD PTR _i$[ebp], ecx
$LN3@main:
        cmp     DWORD PTR _i$[ebp], 10
        jge     SHORT $LN1@main
        mov     edx, DWORD PTR _i$[ebp]
        mov     al, BYTE PTR _i$[ebp]
        mov     BYTE PTR _array$[ebp+edx], al
        jmp     SHORT $LN2@main
$LN1@main:
        mov     ecx, DWORD PTR _fakearray$[ebp]
        ; ecx=adresse de fakearray[0], ecx+1 est fakearray[1] ou array[0]
        movzx   edx, BYTE PTR [ecx+1]
        push    edx
        push    OFFSET $SG2751 ; 'first element %d'
        call    _printf
        add     esp, 8
        mov     eax, DWORD PTR _fakearray$[ebp]
        ; eax=adresse de fakearray[0], eax+2 est fakearray[2] ou array[1]
        movzx   ecx, BYTE PTR [eax+2]
```

```
        push    ecx
        push    OFFSET $SG2752 ; 'second element %d'
        call    _printf
        add     esp, 8
        mov     edx, DWORD PTR _fakearray$[ebp]
        ; edx=adresse de fakearray[0], edx+10 est fakearray[10] ou array[9]
        movzx   eax, BYTE PTR [edx+10]
        push    eax
        push    OFFSET $SG2753 ; 'last element %d'
        call    _printf
        add     esp, 8
        ; soustrait 4, 3, 2 et 1 du pointeur sur array[0] afin de trouver les valeurs avant array[]
        lea     ecx, DWORD PTR _array$[ebp]
        movzx   edx, BYTE PTR [ecx-4]
        push    edx
        lea     eax, DWORD PTR _array$[ebp]
        movzx   ecx, BYTE PTR [eax-3]
        push    ecx
        lea     edx, DWORD PTR _array$[ebp]
        movzx   eax, BYTE PTR [edx-2]
        push    eax
        lea     ecx, DWORD PTR _array$[ebp]
        movzx   edx, BYTE PTR [ecx-1]
        push    edx
        push    OFFSET $SG2754 ; 'array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]=%02X'
        call    _printf
        add     esp, 20
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main   ENDP
```

Donc nous avons le tableau `array[]` de dix éléments, rempli avec les octets $0 \ldots 9$.

Puis nous avons le pointeur `fakearray[]`, qui pointe un octet avant `array[]`.

`fakearray[1]` pointe exactement sur `array[0]`. Mais nous sommes toujours curieux, qu'y a-t-il avant `array[]` ? Nous avons ajouté `random_value` avant `array[]` et l'avons défini à `0x11223344`. Le compilateur sans optimisation a alloué les variables dans l'ordre dans lequel elles sont déclarées, donc oui, la valeur 32-bit `random_value` est juste avant le tableau.

Nous le lançons, et:

```
first element 0
second element 1
last element 9
array[-1]=11, array[-2]=22, array[-3]=33, array[-4]=44
```

Voici le fragment de pile que nous avons copier/coller depuis la fenêtre de pile d'OllyDbg (avec les commentaires ajoutés par l'auteur) :

Listing 3.123 : MSVC 2010 sans optimisation

```
Pile du CPU
Address   Value
001DFBCC  /001DFBD3 ; pointeur fakearray
001DFBD0  |11223344 ; random_value
001DFBD4  |03020100 ; 4 octets de array[]
001DFBD8  |07060504 ; 4 octets de array[]
001DFBDC  |00CB0908 ; reste aléatoire + 2 dernier octets de array[]
001DFBE0  |0000000A ; dernière valeur de i après la fin de la boucle
001DFBE4  |001DFC2C ; valeur de EBP sauvée
001DFBE8  \00CB129D ; Adresse de Retour
```

Le pointeur sur `fakearray[]` (`0x001DFBD3`) est en effet l'adresse de `array[]` dans la pile (`0x001DFBD4`), mais moins 1 octet.

C'est un truc très astucieux et douteux. Personne ne devrait l'utiliser dans du code de production mais comme démonstration, ça joue parfaitement son rôle.

3.23 Plus loin avec les pointeurs

> The way C handles pointers, for example, was a brilliant innovation; it solved a lot of problems that we had before in data structuring and made the programs look good afterwards.
>
> Donald Knuth, interview (1993)

Pour ceux qui veulent se casser la tête à comprendre les pointeurs C/C++, voici plus d'exemples. Certains d'entre eux sont bizarres et ne servent qu'à des fins de démonstration: utilisez-les en production uniquement si vous savez vraiment ce que vous faites.

3.23.1 Travailler avec des adresses au lieu de pointeurs

Un pointeur est juste une adresse en mémoire. Mais pourquoi écrivons-nous `char* string` au lieu de quelque chose comme `address string`? La variable pointeur est fournie avec un type de la valeur sur laquelle le pointeur pointe. Donc le compilateur est capable de détecter des bugs de type de données lors de la compilation.

Pour être pédant, les types de données des langages de programmation ne servent qu'a prévenir des bugs et à l'auto-documentation. Il est possible de n'utiliser que deux types de données, comme *int* (ou *int64_t*) et l'octet—ce sont les seuls types disponible aux programmeurs en langage d'assemblage. Mais c'est une tâche extrêmement difficile d'écrire des programmes en assembleur pratique et sans bugs méchants. La plus petite typo peut conduire à un bug difficile-à-trouver.

L'information sur le type de données est absente d'un code compilé (et c'est l'un des problèmes majeurs pour les dé-compilateurs), et je peux le prouver.

Ceci est ce qu'un programmeur C/C++ peut écrire:

```
#include <stdio.h>
#include <stdint.h>

void print_string (char *s)
{
        printf ("(address: 0x%llx)\n", s);
        printf ("%s\n", s);
};

int main()
{
        char *s="Hello, world!";

        print_string (s);
};
```

Ceci est ce que je peux écrire:

```
#include <stdio.h>
#include <stdint.h>

void print_string (uint64_t address)
{
        printf ("(address: 0x%llx)\n", address);
        puts ((char*)address);
};

int main()
{
        char *s="Hello, world!";

        print_string ((uint64_t)s);
};
```

J'utilise *uint64_t* car j'ai effectué cet exemple sur Linux x64. *int* fonctionnerait pour des OS-s 32-bit. D'abord, un pointeur sur un caractère (la toute première chaîne de bienvenu) est casté en *uint64_t*, puis passé plus loin. La fonction `print_string()` re-caste la valeur *uint64_t* en un pointeur sur un caractère.

Ce qui est intéressant, c'est que GCC 4.8.4 produit une sortie assembleur identique pour les deux versions:

```
gcc 1.c -S -masm=intel -O3 -fno-inline
```

```
.LC0:
        .string "(address: 0x%llx)\n"
print_string:
        push    rbx
        mov     rdx, rdi
        mov     rbx, rdi
        mov     esi, OFFSET FLAT:.LC0
        mov     edi, 1
```

```
        xor     eax, eax
        call    __printf_chk
        mov     rdi, rbx
        pop     rbx
        jmp     puts
.LC1:
        .string "Hello, world!"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC1
        call    print_string
        add     rsp, 8
        ret
```

(j'ai supprimé toutes les directives non significatives de GCC.)

J'ai aussi essayé différents utilitaires UNIX de *diff* et ils ne montrent aucune différence.

Continuons à abuser massivement des traditions de programmation de C/C++. On pourrait écrire ceci:

```
#include <stdio.h>
#include <stdint.h>

uint8_t load_byte_at_address (uint8_t* address)
{
        return *address;
        //this is also possible: return address[0];
};

void print_string (char *s)
{
        char* current_address=s;
        while (1)
        {
                char current_char=load_byte_at_address(current_address);
                if (current_char==0)
                        break;
                printf ("%c", current_char);
                current_address++;
        };
};

int main()
{
        char *s="Hello, world!";

        print_string (s);
};
```

Ça pourrait être récrit comme ceci:

```
#include <stdio.h>
```

```
#include <stdint.h>

uint8_t load_byte_at_address (uint64_t address)
{
        return *(uint8_t*)address;
};

void print_string (uint64_t address)
{
        uint64_t current_address=address;
        while (1)
        {
                char current_char=load_byte_at_address(current_address);
                if (current_char==0)
                        break;
                printf ("%c", current_char);
                current_address++;
        };
};

int main()
{
        char *s="Hello, world!";

        print_string ((uint64_t)s);
};
```

Les deux codes source donnent la même sortie assembleur:

```
gcc 1.c -S -masm=intel -O3 -fno-inline
```

```
load_byte_at_address:
        movzx   eax, BYTE PTR [rdi]
        ret
print_string:
.LFB15:
        push    rbx
        mov     rbx, rdi
        jmp     .L4
.L7:
        movsx   edi, al
        add     rbx, 1
        call    putchar
.L4:
        mov     rdi, rbx
        call    load_byte_at_address
        test    al, al
        jne     .L7
        pop     rbx
        ret
.LC0:
        .string "Hello, world!"
main:
```

```
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0
        call    print_string
        add     rsp, 8
        ret
```

(j'ai supprimé toutes les directives non significatives de GCC.)

Aucune différence: les pointeurs C/C++ sont essentiellement des adresses, mais fournies avec une information sur le type, afin de prévenir des erreurs possible lors de la compilation.

3.23.2 Passer des valeurs en tant que pointeurs; tagged unions

Voici un exemple montrant comment passer des valeurs dans des pointeurs:

```
#include <stdio.h>
#include <stdint.h>

uint64_t multiply1 (uint64_t a, uint64_t b)
{
        return a*b;
};

uint64_t* multiply2 (uint64_t *a, uint64_t *b)
{
        return (uint64_t*)((uint64_t)a*(uint64_t)b);
};

int main()
{
        printf ("%d\n", multiply1(123, 456));
        printf ("%d\n", (uint64_t)multiply2((uint64_t*)123, (uint64_t*)456)↙
    ↪ );
};
```

Il fonctionne sans problème et GCC 4.8.4 compile les fonctions multiply1() et multiply2() de manière identique!

```
multiply1:
        mov     rax, rdi
        imul    rax, rsi
        ret

multiply2:
        mov     rax, rdi
        imul    rax, rsi
        ret
```

Tant que vous ne déréférencez pas le pointeur (autrement dit, que vous ne lisez aucune donnée depuis l'adresse stockée dans le pointeur), tout se passera bien.

Un pointeur est une variable qui peut stocker n'importe quoi, comme une variable usuelle.

L'instruction de multiplication signée (IMUL) est utilisée ici au lieu de la non-signée (MUL), lisez-en plus à ce sujet ici: 11.1 on page 1287.

À propos, il y a une astuce très connue pour abuser des pointeurs appelée *tagged pointers*. En gros, si tous vos pointeurs pointent sur des blocs de mémoire de taille, disons, 16 octets (ou qu'ils sont toujours alignés sur une limite de 16-octet), les 4 bits les plus bas du pointeur sont toujours zéro et cet espace peut être utilisé d'une certaine façon. C'est très répandu dans les compilateurs et interpréteurs LISP. Ils stockent le type de cell/objet dans ces bits inutilisés, ceci peut économiser un peu de mémoire. Encore mieux, vous pouvez évaluer le type de cell/objet en utilisant seulement le pointeur, sans accès supplémentaire à la mémoire. En lire plus à ce sujet: [Dennis Yurichev, *C/C++ programming language notes* 1.3].

3.23.3 Abus de pointeurs dans le noyau Windows

La section ressource d'un exécutable PE dans les OS Windows est une section contenant des images, des icônes, des chaînes, etc. Les premières versions de Windows permettaient seulement d'adresser les ressources par ID, mais Microsoft a ajouté un moyen de les adresser en utilisant des chaînes.

Donc, il doit être alors possible de passer un ID ou une chaîne a la fonction Elle est déclarée comme ceci: FindResource().

```
HRSRC WINAPI FindResource(
  _In_opt_ HMODULE hModule,
  _In_     LPCTSTR lpName,
  _In_     LPCTSTR lpType
);
```

lpName et *lpType* ont un type *char** ou *wchar**, et lorsque quelqu'un veut encore passer un ID, il doit utiliser la macro MAKEINTRESOURCE, comme ceci:

```
result = FindResource(..., MAKEINTRESOURCE(1234), ...);
```

C'est un fait intéressant que MAKEINTRESOURCE est juste un casting d'entier vers un pointeur. Dans MSVC 2013, dans le fichier
Microsoft SDKs\Windows\v7.1A\Include\Ks.h nous pouvons voir ceci:

```
...

#if (!defined( MAKEINTRESOURCE ))
#define MAKEINTRESOURCE( res ) ((ULONG_PTR) (USHORT) res)
#endif

...
```

Ça semble fou. Regardons dans l'ancien code source de Windows NT4 qui avait fuité. Dans *private/windows/base/client/module.c* nous pouvons trouver le source code de *FindResource()* :

```
HRSRC
FindResourceA(
    HMODULE hModule,
    LPCSTR lpName,
    LPCSTR lpType
    )

...

{
    NTSTATUS Status;
    ULONG IdPath[ 3 ];
    PVOID p;

    IdPath[ 0 ] = 0;
    IdPath[ 1 ] = 0;
    try {
        if ((IdPath[ 0 ] = BaseDllMapResourceIdA( lpType )) == -1) {
            Status = STATUS_INVALID_PARAMETER;
            }
        else
        if ((IdPath[ 1 ] = BaseDllMapResourceIdA( lpName )) == -1) {
            Status = STATUS_INVALID_PARAMETER;
...
```

Continuons avec *BaseDllMapResourceIdA()* dans le même fichier source:

```
ULONG
BaseDllMapResourceIdA(
    LPCSTR lpId
    )
{
    NTSTATUS Status;
    ULONG Id;
    UNICODE_STRING UnicodeString;
    ANSI_STRING AnsiString;
    PWSTR s;

    try {
        if ((ULONG)lpId & LDR_RESOURCE_ID_NAME_MASK) {
            if (*lpId == '#') {
                Status = RtlCharToInteger( lpId+1, 10, &Id );
                if (!NT_SUCCESS( Status ) || Id & LDR_RESOURCE_ID_NAME_MASK↙
    ↘ ) {
                    if (NT_SUCCESS( Status )) {
                        Status = STATUS_INVALID_PARAMETER;
                        }
                    BaseSetLastNTError( Status );
                    Id = (ULONG)-1;
                    }
                }
            else {
                RtlInitAnsiString( &AnsiString, lpId );
```

```
                Status = RtlAnsiStringToUnicodeString( &UnicodeString,
                                                      &AnsiString,
                                                      TRUE
                                                    );
                if (!NT_SUCCESS( Status )){
                    BaseSetLastNTError( Status );
                    Id = (ULONG)-1;
                    }
                else {
                    s = UnicodeString.Buffer;
                    while (*s != UNICODE_NULL) {
                        *s = RtlUpcaseUnicodeChar( *s );
                        s++;
                        }

                    Id = (ULONG)UnicodeString.Buffer;
                    }
                }
            }
        else {
            Id = (ULONG)lpId;
            }
        }
    except (EXCEPTION_EXECUTE_HANDLER) {
        BaseSetLastNTError( GetExceptionCode() );
        Id =  (ULONG)-1;
        }
    return Id;
}
```

lpId est ANDé avec *LDR_RESOURCE_ID_NAME_MASK*.
Que nous pouvons trouver dans *public/sdk/inc/ntldr.h* :

```
...

#define LDR_RESOURCE_ID_NAME_MASK 0xFFFF0000

...
```

Donc *lpId* est ANDé avec *0xFFFF0000* et si des bits sont encore présents dans la partie 16-bit basse, la première partie de la fonction est exécutée (*lpId* est traité comme une adresse de chaîne). Autrement—la seconde moitié (*lpId* est traitée comme une valeur 16-bit).

Encore, ce code peut être trouvé dans le fichier kernel32.dll de Windows 7:

```
....

.text:0000000078D24510 ;
    __int64 __fastcall BaseDllMapResourceIdA(PCSZ SourceString)
.text:0000000078D24510 BaseDllMapResourceIdA proc near        ; CODE XREF:
    FindResourceExA+34
.text:0000000078D24510                                        ;
    FindResourceExA+4B
```

```
.text:0000000078D24510
.text:0000000078D24510 var_38          = qword ptr -38h
.text:0000000078D24510 var_30          = qword ptr -30h
.text:0000000078D24510 var_28          = _UNICODE_STRING ptr -28h
.text:0000000078D24510 DestinationString= _STRING ptr -18h
.text:0000000078D24510 arg_8           = dword ptr  10h
.text:0000000078D24510
.text:0000000078D24510 ; FUNCTION CHUNK AT .text:0000000078D42FB4 SIZE
    000000D5 BYTES
.text:0000000078D24510
.text:0000000078D24510                 push    rbx
.text:0000000078D24512                 sub     rsp, 50h
.text:0000000078D24516                 cmp     rcx, 10000h
.text:0000000078D2451D                 jnb     loc_78D42FB4
.text:0000000078D24523                 mov     [rsp+58h+var_38], rcx
.text:0000000078D24528                 jmp     short $+2
.text:0000000078D2452A ;
    ---------------------------------------------------------------------------
.text:0000000078D2452A
.text:0000000078D2452A loc_78D2452A:                           ; CODE XREF:
    BaseDllMapResourceIdA+18
.text:0000000078D2452A                                         ;
    BaseDllMapResourceIdA+1EAD0
.text:0000000078D2452A                 jmp     short $+2
.text:0000000078D2452C ;
    ---------------------------------------------------------------------------
.text:0000000078D2452C
.text:0000000078D2452C loc_78D2452C:                           ;
    CODE XREF: BaseDllMapResourceIdA:loc_78D2452A
.text:0000000078D2452C                                         ;
    BaseDllMapResourceIdA+1EB74
.text:0000000078D2452C                 mov     rax, rcx
.text:0000000078D2452F                 add     rsp, 50h
.text:0000000078D24533                 pop     rbx
.text:0000000078D24534                 retn
.text:0000000078D24534 ;
    ---------------------------------------------------------------------------
.text:0000000078D24535                 align 20h
.text:0000000078D24535 BaseDllMapResourceIdA endp


....

.text:0000000078D42FB4 loc_78D42FB4:                           ; CODE XREF:
    BaseDllMapResourceIdA+D
.text:0000000078D42FB4                 cmp     byte ptr [rcx], '#'
.text:0000000078D42FB7                 jnz     short loc_78D43005
.text:0000000078D42FB9                 inc     rcx
.text:0000000078D42FBC                 lea     r8, [rsp+58h+arg_8]
.text:0000000078D42FC1                 mov     edx, 0Ah
.text:0000000078D42FC6                 call    cs:__imp_RtlCharToInteger
.text:0000000078D42FCC                 mov     ecx, [rsp+58h+arg_8]
.text:0000000078D42FD0                 mov     [rsp+58h+var_38], rcx
.text:0000000078D42FD5                 test    eax, eax
.text:0000000078D42FD7                 js      short loc_78D42FE6
.text:0000000078D42FD9                 test    rcx, 0FFFFFFFFFFFF0000h
.text:0000000078D42FE0                 jz      loc_78D2452A
```

```
....
```

Si la valeur du pointeur en entrée est plus grande que 0x10000, un saut au traitement de chaînes se produit. Autrement, la valeur en entrée du *lpId* est renvoyée telle quelle. Le masque *0xFFFF0000* n'est plus utilisé ici, car ceci est du code 64-bit, mais encore, *0xFFFFFFFFFFFF0000* pourrait fonctionner ici.

Le lecteur attentif pourrait demander ce qui se passe si l'adresse de la chaîne en entrée est plus petite que 0x10000? Ce code se base sur le fait que dans Windows, il n'y a rien aux adresses en dessous de 0x10000, au moins en Win32 realm.

Raymond Chen écrit à propos de ceci:

> How does MAKEINTRESOURCE work? It just stashes the integer in the bottom 16 bits of a pointer, leaving the upper bits zero. This relies on the convention that the first 64KB of address space is never mapped to valid memory, a convention that is enforced starting in Windows 7.

En quelques mots, ceci est un sale hack et probablement qu'il ne devrait être utilisé qu'en cas de réelle nécessité. Peut-être que la fonction *FindResource()* avait un type *SHORT* pour ses arguments, et puis Microsoft a ajouté un moyen de passer des chaînes ici, mais que le code ancien devait toujours être supporté.

Maintenant, voici un exemple distillé:

```
#include <stdio.h>
#include <stdint.h>

void f(char* a)
{
        if (((uint64_t)a)>0x10000)
                printf ("Pointer to string has been passed: %s\n", a);
        else
                printf ("16-bit value has been passed: %d\n", (uint64_t)a);
};

int main()
{
        f("Hello!"); // pass string
        f((char*)1234); // pass 16-bit value
};
```

Ça fonctionne!

Abus de pointeurs dans le noyau Linux

Comme ça a déjà été pointé dans des commentaires sur Hacker News, le noyau Linux comporte aussi des choses comme ça.

Par exemple, cette fonction peut renvoyer un code erreur ou un pointeur:

```
struct kernfs_node *kernfs_create_link(struct kernfs_node *parent,
                                       const char *name,
                                       struct kernfs_node *target)
{
        struct kernfs_node *kn;
        int error;

        kn = kernfs_new_node(parent, name, S_IFLNK|S_IRWXUGO, KERNFS_LINK);
        if (!kn)
                return ERR_PTR(-ENOMEM);

        if (kernfs_ns_enabled(parent))
                kn->ns = target->ns;
        kn->symlink.target_kn = target;
        kernfs_get(target);     /* ref owned by symlink */

        error = kernfs_add_one(kn);
        if (!error)
                return kn;

        kernfs_put(kn);
        return ERR_PTR(error);
}
```

(https://github.com/torvalds/linux/blob/fceef393a538134f03b778c5d2519e670269342f/fs/kernfs/symlink.c#L25)

ERR_PTR est une macro pour caster un entier en un pointeur:

```
static inline void * __must_check ERR_PTR(long error)
{
        return (void *) error;
}
```

(https://github.com/torvalds/linux/blob/61d0b5a4b2777dcf5daef245e212b3c1fa8091ca/tools/virtio/linux/err.h)

Ce fichier d'en-tête contient aussi une macro d'aide pour distinguer un code d'erreur d'un pointeur:

```
#define IS_ERR_VALUE(x) unlikely((x) >= (unsigned long)-MAX_ERRNO)
```

Ceci signifie que les codes erreurs sont les "pointeurs" qui sont très proche de -1, et, heureusement, il n'y a rien dans la mémoire du noyau à des adresses comme 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFE, 0xFFFFFFFFFFFFFFFD, etc.

Une méthode bien plus répandue est de renvoyer *NULL* en cas d'erreur et de passer le code d'erreur par un argument supplémentaire. Les auteurs du noyau Linux ne font pas ça, mais quiconque veut utiliser ces fonctions doit toujours garder en mémoire que le pointeur renvoyé doit toujours être testé avec *IS_ERR_VALUE* avant d'être déréférencé.

Par exemple:

```
        fman->cam_offset = fman_muram_alloc(fman->muram, fman->cam_size);
        if (IS_ERR_VALUE(fman->cam_offset)) {
                dev_err(fman->dev, "%s: MURAM alloc for DMA CAM failed\n",
                        __func__);
                return -ENOMEM;
        }
```

(https://github.com/torvalds/linux/blob/aa00edc1287a693eadc7bc67a3d73555d969b35d/drivers/net/ethernet/freescale/fman/fman.c#L826)

Abus de pointeurs dans l'espace utilisateur UNIX

La fonction mmap() renvoie -1 en cas d'erreur (ou MAP_FAILED, qui vaut -1). Certaines personnes disent que mmap() peut mapper une zone mémoire à l'adresse zéro dans de rares situations, donc elle ne peut pas utiliser 0 ou NULL comme code d'erreur.

3.23.4 Pointeurs nuls

"Null pointer assignment" erreur du temps de MS-DOS

Des anciens peuvent se souvenir d'un message d'erreur bizarre du temps de MS-DOS: "Null pointer assignment". Qu'est-ce que ça signifie?

Il n'est pas possible d'écrire à l'adresse mémoire zéro avec les OSs *NIX et Windows, mais il est possible de le faire avec MS-DOS, à cause de l'absence de protection de la mémoire.

Donc, j'ai sorti mon ancien Turbo C++ 3.0 (qui fût renommer plus tard en Borland C++) d'avant les années 1990s et essayé de compiler ceci:

```
#include <stdio.h>

int main()
{
        int *ptr=NULL;
        *ptr=1234;
        printf ("Now let's read at NULL\n");
        printf ("%d\n", *ptr);
};
```

C'est difficile à croire, mais ça fonctionne, sans erreur jusqu'à la sortie, toutefois:

Listing 3.124 : Ancient Turbo C 3.0

```
C:\TC30\BIN\1
Now let's read at NULL
1234
Null pointer assignment

C:\TC30\BIN>_
```

Plongeons un peu plus profondément dans le code du CRT de Borland C++ 3.1, fichier *c0.asm* :

```
; _checknull()    check for null pointer zapping copyright message

...

; Check for null pointers before exit

__checknull     PROC    DIST
                PUBLIC  __checknull

IF      LDATA  EQ  false
  IFNDEF  __TINY__
                push    si
                push    di
                mov     es, cs:DGROUP@@
                xor     ax, ax
                mov     si, ax
                mov     cx, lgth_CopyRight
ComputeChecksum label   near
                add     al, es:[si]
                adc     ah, 0
                inc     si
                loop    ComputeChecksum
                sub     ax, CheckSum
                jz      @@SumOK
                mov     cx, lgth_NullCheck
                mov     dx, offset DGROUP: NullCheck
                call    ErrorDisplay
@@SumOK:        pop     di
                pop     si
  ENDIF
ENDIF

_DATA           SEGMENT

; Magic symbol used by the debug info to locate the data segment
                public DATASEG@
DATASEG@        label   byte

; The CopyRight string must NOT be moved or changed without
; changing the null pointer check logic

CopyRight       db      4 dup(0)
                db      'Borland C++ - Copyright 1991 Borland Intl.',0
lgth_CopyRight  equ     $ - CopyRight

IF      LDATA  EQ  false
IFNDEF  __TINY__
CheckSum        equ     00D5Ch
NullCheck       db      'Null pointer assignment', 13, 10
lgth_NullCheck  equ     $ - NullCheck
ENDIF
```

```
ENDIF

...
```

Le modèle de mémoire de MS-DOS était vraiment bizarre (11.7 on page 1297) et ne vaut probablement pas la peine de s'y plonger, à moins d'être fan de rétro-computing ou de rétro-gaming. Une chose que nous devons garder à l'esprit est que le segment de mémoire (segment de données inclus) dans MS-DOS, est un segment de mémoire dans lequel du code ou des données sont stockés, mais contrairement au OSs “sérieux”, il commence à l'adresse 0.

Et dans le CRT de Borland C++, le segment de données commence avec 4 octets à zéro puis la chaîne de copyright “Borland C++ - Copyright 1991 Borland Intl.”. L'intégrité des 4 octets à zéro et de la chaîne de texte est vérifiée en sortant, et s'ils sont corrompus, le message d'erreur est affiché.

Mais pourquoi? Écrire à l'adresse zéro est une erreur courante en C/C++, et si vous faites cela sur *NIX ou Windows, votre application va planter. MS-DOS n'a pas de protection de la mémoire, donc le CRT doit vérifier ceci post-factum et le signaler à la sortie. Si vous voyez ce message, ceci signifie que votre programme à un certain point, a écrit à l'adresse 0.

Notre programme le fait. Et ceci est pourquoi le nombre 1234 a été lu correctement: car il a été écrit à la place des 4 premiers octets à zéro. La somme de contrôle est incorrecte à la sortie (car le nombre y a été laissé), donc le message a été affiché.

Ai-je raison? J'ai récrit le programme pour vérifier mes suppositions:

```
#include <stdio.h>

int main()
{
        int *ptr=NULL;
        *ptr=1234;
        printf ("Now let's read at NULL\n");
        printf ("%d\n", *ptr);
        *ptr=0; // psst, cover our tracks!
};
```

Ce programme s'exécute sans message d'erreur à la sortie.

Une telle méthode pour avertir en cas d'assignation du pointeur nul est pertinente pour MS-DOS, peut-être qu'elle peut encore être utilisée de nos jours avec des MCUs à bas coût sans protection de la mémoire et/ou MMU[37].

Pourquoi voudrait-on écrire à l'adresse 0?

Mais pourquoi un programmeur sain d'esprit écrirait du code écrivant quelque chose à l'adresse 0? Ça peut être fait accidentellement: par exemple, un pointeur doit être initialisé au bloc de mémoire nouvellement alloué et ensuite passé à quelque fonction qui renvoie des données à travers un pointeur.

[37]Memory Management Unit

```
int *ptr=NULL;

\dots nous avons oublié d'allouer la mémoire et d'initialiser ptr

strcpy (ptr, buf); // strcpy() termine silencieusement car MS-DOS n'a pas de
    protection de la mémoire
```

Encore pire:

```
int *ptr=malloc(1000);

\dots nous avons oublié de vérifier si la mémoire a réellement été allouée:
    ↳ c'est MS-DOS après tout et les ordinateurs ont une petite quantité
    ↳ de RAM,
\dots et être à court de RAM était très courant.
\dost si malloc() a renvoyé NULL, ptr sera aussi NULL.

strcpy (ptr, buf); // strcpy() termine silencieusement car MS-DOS n'a pas
    ↳ de protection de la mémoire
```

Écrire sciemment à l'adresse 0

Voici un exemple tiré de dmalloc[38], une façon portable de générer un core dump, si les autres moyens ne sont pas disponibles:

```
3.4 Generating a Core File on Errors
====================================

If the `error-abort' debug token has been enabled, when the library
detects any problems with the heap memory, it will immediately attempt
to dump a core file.  *Note Debug Tokens::.  Core files are a complete
copy of the program and it's state and can be used by a debugger to see
specifically what is going on when the error occurred.  *Note Using
With a Debugger::.  By default, the low, medium, and high arguments to
the library utility enable the `error-abort' token.  You can disable
this feature by entering `dmalloc -m error-abort' (-m for minus) to
remove the `error-abort' token and your program will just log errors
and continue.  You can also use the `error-dump' token which tries to
dump core when it sees an error but still continue running.  *Note
Debug Tokens::.

   When a program dumps core, the system writes the program and all of
its memory to a file on disk usually named `core'.  If your program is
called `foo' then your system may dump core as `foo.core'.  If you are
not getting a `core' file, make sure that your program has not changed
to a new directory meaning that it may have written the core file in a
different location.  Also insure that your program has write privileges
over the directory that it is in otherwise it will not be able to dump
a core file.  Core dumps are often security problems since they contain
all program memory so systems often block their being produced.  You
```

[38] http://dmalloc.com/

```
will want to check your user and system's core dump size ulimit
settings.

   The library by default uses the `abort' function to dump core which
may or may not work depending on your operating system.  If the
following program does not dump core then this may be the problem.  See
`KILL_PROCESS' definition in `settings.dist'.

     main()
     {
       abort();
     }

   If `abort' does work then you may want to try the following setting
in `settings.dist'.  This code tries to generate a segmentation fault
by dereferencing a `NULL' pointer.

     #define KILL_PROCESS    { int *_int_p = 0L; *_int_p = 1; }
```

NULL en C/C++

NULL en C/C++ est juste une macro qui est souvent définie comme ceci:

```
#define NULL  ((void*)0)
```

(fichier libio.h)

*void** est un type de données reflétant le fait que c'est un pointeur, mais sur une valeur d'un type de données inconnu (*void*).

NULL est usuellement utilisé pour montrer l'absence d'un objet. Par exemple, vous avez une liste simplement chaînée, et chaque nœud a une valeur (ou un pointeur sur une valeur) et un pointeur *next*. Pour montrer qu'il n'y a pas de nœud suivant, 0 est stocké dans le champ *next*. (D'autres solutions sont pires.) Peut-être pourriez-vous avoir un environnement fou où vous devriez allouer un bloc de mémoire à l'adresse zéro. Comment indiqueriez-vous l'absence de nœud suivant? Avec une sorte de *magic number*? Peut-être -1? Ou peut-être avec un bit additionnel?

Nous trouvons ceci dans Wikipédia:

> In fact, quite contrary to the zero page's original preferential use, some modern operating systems such as FreeBSD, Linux and Microsoft Windows[2] actually make the zero page inaccessible to trap uses of NULL pointers.

(`https://en.wikipedia.org/wiki/Zero_page`)

Pointeur nul à une fonction

Il est possible d'appeler une fonction avec son adresse. Par exemple, je compile ceci avec MSVC 2010 et le lance dans Windows 7:

```
#include <windows.h>
#include <stdio.h>

int main()
{
        printf ("0x%x\n", &MessageBoxA);
};
```

Le résultat est *0x7578feae* et ne change pas après plusieurs lancements, car user32.dll (où la fonction MessageBoxA se trouve) est toujours chargée à la même adresse. Et aussi car ASLR[39] n'est pas activé (le résultat serait différent à chaque exécution dans ce cas).

Appelons *MessageBoxA()* par son adresse:

```
#include <windows.h>
#include <stdio.h>

typedef int (*msgboxtype)(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption,   ⤦
    ↳ UINT uType);

int main()
{
        msgboxtype msgboxaddr=0x7578feae;

        // force to load DLL into process memory,
        // since our code doesn't use any function from user32.dll,
        // and DLL is not imported
        LoadLibrary ("user32.dll");

        msgboxaddr(NULL, "Hello, world!", "hello", MB_OK);
};
```

Bizarre, mais ça fonctionne avec Windows 7 x86.

Ceci est communément utilisé dans les shellcodes, car il est difficile d'y appeler des fonctions DLL par leur nom. Et ASLR est une contre-mesure.

Maintenant, ce qui est vraiment bizarre, quelques programmeurs C embarqué peuvent être familiers avec un code comme ceci:

```
int reset()
{
        void (*foo)(void) = 0;
        foo();
};
```

Qui voudrait appeler une fonction à l'adresse 0? Ceci est un moyen portable de sauter à l'adresse zéro. De nombreux micro-contrôleurs à bas coût n'ont pas de protection mémoire ou de MMU et après un reset, ils commencent à exécuter le code à l'adresse 0, où une sorte de code d'initialisation est stocké. Donc sauter à

39Address Space Layout Randomization

l'adresse 0 est un moyen de se réinitialiser. On pourrait utiliser de l'assembleur inline, mais si ce n'est pas possible, cette méthode portable est utilisable.

Ça compile même correctement avec mon GCC 4.8.4 sur Linux x64:

```
reset:
        sub     rsp, 8
        xor     eax, eax
        call    rax
        add     rsp, 8
        ret
```

Le fait que le pointeur de pile soit décalé n'est pas un problème: le code d'initialisation dans les micro-contrôleurs ignorent en général les registres et l'état de la RAM et démarrent from scratch.

Et bien sûr, ce code planterait sur *NIX ou Windows à cause de la protection mémoire, et même sans la protection mémoire, il n'y a pas de code à l'adresse 0.

GCC possède même des extensions non-standard, permettant de sauter à une adresse spécifique plutôt qu'un appel à une fonction ici: `http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html`.

3.23.5 Tableaux comme argument de fonction

On peut se demander quelle est la différence entre déclarer le type d'un argument de fonction en tant que tableau et en tant que pointeur?

Il semble qu' il n'y ai pas du tout de différence:

```
void write_something1(int a[16])
{
        a[5]=0;
};

void write_something2(int *a)
{
        a[5]=0;
};

int f()
{
        int a[16];
        write_something1(a);
        write_something2(a);
};
```

GCC 4.8.4 avec optimisation :

```
write_something1:
        mov     DWORD PTR [rdi+20], 0
        ret

write_something2:
```

```
        mov     DWORD PTR [rdi+20], 0
        ret
```

Mais vous pouvez toujours déclarer un tableau au lieu d'un pointeur à des fins d'auto-documentation, si la taille du tableau est toujours fixée. Et peut-être, des outils d'analyse statique seraient capable de vous avertir d'un possible débordement de tampon. Ou est-ce possible avec des outils aujourd'hui?

Certaines personnes, incluant Linux Torvalds, critiquent cette possibilité de C/C++ : https://lkml.org/lkml/2015/9/3/428.

Le standard C99 a aussi le mot-clef *static* [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007) 6.7.5.3] :

> If the keyword static also appears within the [and] of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.

Si le mot-clef static apparaît entre les [et] du tableau de dérivation de type, alors pour chaque appel à la fonction,

3.23.6 Pointeur sur une fonction

Un nom de fonction en C/C++ sans parenthèses, comme "printf" est un pointeur sur une fonction du type *void (*)()*. Essayons de lire le contenu de la fonction et de la patcher:

```
#include <memory.h>
#include <stdio.h>

void print_something ()
{
        printf ("we are in %s()\n", __FUNCTION__);
};

int main()
{
        print_something();
        printf ("first 3 bytes: %x %x %x...\n",
                        *(unsigned char*)print_something,
                        *((unsigned char*)print_something+1),
                        *((unsigned char*)print_something+2));

        *(unsigned char*)print_something=0xC3; // RET's opcode
        printf ("going to call patched print_something():\n");
        print_something();
        printf ("it must exit at this point\n");

};
```

Ça dit que les 3 premiers octets de la fonction sont 55 89 e5. En effet, ce sont les opcodes des instructions PUSH EBP et MOV EBP, ESP (se sont des opcodes x86). Mais alors notre programme plante, car la section *text* est en lecture seule.

Nous pouvons recompiler notre exemple et rendre la section *text* modifiable [40] :

```
gcc --static -g -Wl,--omagic -o example example.c
```

Ça fonctionne!

```
we are in print_something()
first 3 bytes: 55 89 e5...
going to call patched print_something():
it must exit at this point
```

3.23.7 Pointeur sur une fonction: protection contre la copie

Un pirate de logiciel peut trouver une fonction qui vérifie la protection et renvoie *vrai* ou *faux*.

Peut-on vérifier son intégrité? Il s'avère que cela peut être fait facilement.

D'après objdump, les 3 premiers octets de check_protection() sont 0x55 0x89 0xE5 (compte tenu du fait qu'il s'agit de GCC sans optimisation) :

```
#include <stdlib.h>
#include <stdio.h>

int check_protection()
{
        // do something
        return 0;
        // or return 1;
};

int main()
{
        if (check_protection()==0)
        {
                printf ("no protection installed\n");
                exit(0);
        };

        // ...and then, at some very important point...
        if (*(((unsigned char*)check_protection)+0) != 0x55)
        {
                printf ("1st byte has been altered\n");
                // do something mean, add watermark, etc
        };
        if (*(((unsigned char*)check_protection)+1) != 0x89)
        {
                printf ("2nd byte has been altered\n");
```

[40] http://stackoverflow.com/questions/27581279/make-text-segment-writable-elf

```
                // do something mean, add watermark, etc
        };
        if (*(((unsigned char*)check_protection)+2) != 0xe5)
        {
                printf ("3rd byte has been altered\n");
                // do something mean, add watermark, etc
        };
};
```

```
0000054d <check_protection>:
 54d:   55                      push   %ebp
 54e:   89 e5                   mov    %esp,%ebp
 550:   e8 b7 00 00 00          call   60c <__x86.get_pc_thunk.ax>
 555:   05 7f 1a 00 00          add    $0x1a7f,%eax
 55a:   b8 00 00 00 00          mov    $0x0,%eax
 55f:   5d                      pop    %ebp
 560:   c3                      ret
```

Si quelqu'un patchait `check_protection()`, votre programme peut faire quelque chose de méchant, peut-être se terminer brusquement. (tracer possède l'option BPMx pour ça.)

3.23.8 Pointeur sur une fonction: un bogue courant (ou une typo)

Un bogue/une typo notoire:

```
int expired()
{
        // check license key, current date/time, etc
};

int main()
{
        if (expired) // must be expired() here
        {
                print ("expired\n");
                exit(0);
        }
        else
        {
                // do something
        };
};
```

Puisque le nom de la fonction seul est interprété comme un pointeur sur une fonction, ou une adresse, la déclaration `if(function_name)` est comme `if(true)`.

Malheureusement, un compilateur C/C++ ne va pas générer d'avertissement.

3.23.9 Pointeur comme un identificateur d'objet

Tant le langage assembleur que le C n'ont pas de fonctionnalité POO, mais il est possible d'écrire du code dans un style POO (il suffit de traiter une structure comme un objet).

Il est intéressant que, parfois, un pointeur sur un objet (ou son adresse) soit appelé comme ID (dans le sens de cacher/encapsuler des données).

Par exemple, LoadLibrary(), d'après MSDN[41], renvoie une "handle to the module" [42]. Puis, vous passez ce "handle" à une autre fonction comme GetProcAddress(). Mais en fait, LoadLibrary() renvoie un pointeur sur le fichier DLL mappé en mémoire. [43]. Vous pouvez lire deux octets de l'adresse renvoyée par LoadLibrary(), et ça sera "MZ" (deux premiers octets de n'importe quel fichier .EXE/.DLL sur Windows).

Il semble que Microsoft "cache" cela, afin de fournir une meilleure compatibilité ascendante. Donc, le type de données de HMODULE et HINSTANCE a une autre signification dan Windows 16-bits.

Probablement que ceci est la raison pour laquelle `printf()` possède le modificateur "%p", qui est utilisé pour afficher des pointeurs (entier 32-bits sur une architecture 32-bits et 64-bit sur une 64-bits, etc.) au format hexadécimal. L'adresse d'une structure écrite dans des logs de debug peut aider à la retrouver dans d'autres logs.

Voici un exemple tiré du code source de SQLite:

```
...

struct Pager {
  sqlite3_vfs *pVfs;          /* OS functions to use for IO */
  u8 exclusiveMode;           /* Boolean. True if locking_mode==EXCLUSIVE ⤦
    ↳ */
  u8 journalMode;             /* One of the PAGER_JOURNALMODE_* values */
  u8 useJournal;              /* Use a rollback journal on this file */
  u8 noSync;                  /* Do not sync the journal if true */

....

static int pagerLockDb(Pager *pPager, int eLock){
  int rc = SQLITE_OK;

  assert( eLock==SHARED_LOCK || eLock==RESERVED_LOCK || eLock==⤦
    ↳ EXCLUSIVE_LOCK );
  if( pPager->eLock<eLock || pPager->eLock==UNKNOWN_LOCK ){
    rc = sqlite3OsLock(pPager->fd, eLock);
    if( rc==SQLITE_OK && (pPager->eLock!=UNKNOWN_LOCK||eLock==⤦
    ↳ EXCLUSIVE_LOCK) ){
      pPager->eLock = (u8)eLock;
      IOTRACE(("LOCK %p %d\n", pPager, eLock))
    }
```

[41] Microsoft Developer Network
[42] https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms684175(v=vs.85).aspx
[43] https://blogs.msdn.microsoft.com/oldnewthing/20041025-00/?p=37483

```
  }
  return rc;
}

...

  PAGER_INCR(sqlite3_pager_readdb_count);
  PAGER_INCR(pPager->nRead);
  IOTRACE(("PGIN %p %d\n", pPager, pgno));
  PAGERTRACE(("FETCH %d page %d hash(%08x)\n",
               PAGERID(pPager), pgno, pager_pagehash(pPg)));

...
```

3.23.10 Oracle RDBMS et un simple ramasse miette pour C/C++

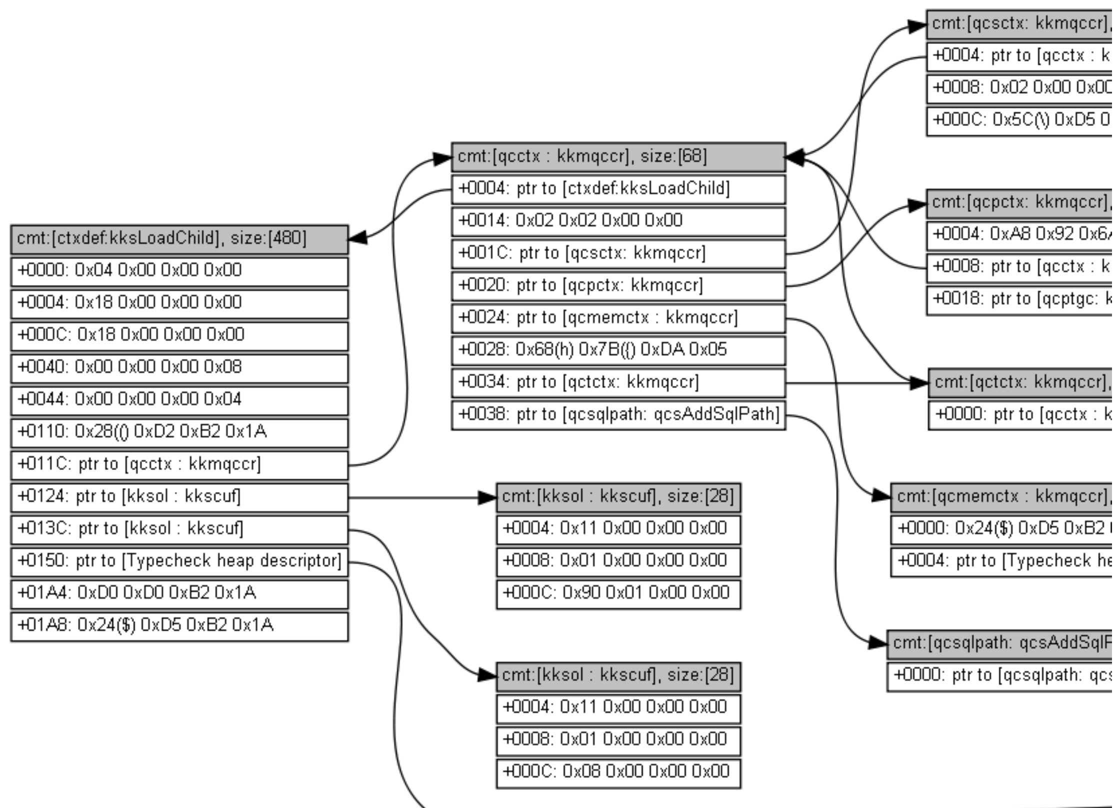
Il fût un temps où j'essayais d'en apprendre plus sur Oracle RDBMS, cherchais des vulnérabilités, etc. C'est un énorme logiciel, et une fonction typique peut prendre de très larges objets imbriqués comme arguments. Et je voulais afficher ces objets, sous forme d'arbres (ou de graphes).

Je suivais aussi toutes les allocations/libérations de mémoire en interceptant les fonctions d'allocation/libération. Et lorsqu'une fonction interceptée prenait un pointeur sur un bloc de mémoire, je cherchais ce bloc dans une liste de blocs alloués. J'obtenais sa taille + un nom court du bloc (ceci est comme "tagué" dans le noyau de l'OS Windows[44]).

Pour un bloc donné, je peux le balayer à la recherche de mots 32-bit (sur les OS 32-bit) ou de mots 64-bit (sur les OS 64-bit). Chaque mot peut être un pointeur sur un autre bloc. Et si c'est le cas (je trouve ceci dans un autre bloc dans mes enregistrements), je peux chercher récursivement.

Et ensuite, en utilisant GraphViz, je peux générer un tel diagramme:

[44] Plus d'information sur les commentaires dans les blocs alloués: Dennis Yurichev, *C/C++ programming language notes* http://yurichev.com/C-book.html



Images plus grosses: 1, 2.

Ceci est assez impressionnant, compte tenu du fait que je n'ai aucune information à propos des types de données de toutes ces structures. Mais je peux en obtenir des informations.

Maintenant le ramasse miette pour C/C++: Boehm GC

Si vous utilisez un bloc alloué en mémoire, son adresse doit être présente quelque part, comme un pointeur dans une structure ou un tableau dans un autre bloc alloué, ou dans une structure allouée globale, ou dans une variable locale sur la pile. S'il n'y a plus de pointeur sur un bloc, vous pouvez l'appeler "orphelin", et il est une cause des fuites de mémoire.

Et c'est ce qu'un GC[45] fait. Il balaye tous les blocs (car il garde un œil sur tous les blocs alloués) à la recherche de pointeurs. Il est important de comprendre qu'il n'a aucune idée du type de données de tous les champs de ces structures dans le blocs—ceci est important, le GC n'a aucune information sur les types. Il balayage juste les blocs à la recherche de mots 32-bit ou 64-bit, et regarde s'ils peuvent être des pointeurs sur d'autres bloc(s). Il balaye aussi la pile. Il traite les blocs alloués et la pile comme des tableaux de mots, dont certains pourraient être des pointeurs.

[45] Garbage Collector

Et s'il trouve un bloc alloué, qui est "orphelin", i.e., sur lequel aucun autre pointeur sur lui depuis un autre bloc ou la pile, ce bloc est considéré comme inutile, devant être libéré. Le processus de balayage prend du temps, et c'est pourquoi les GCs sont critiqués.

Ainsi, un GC comme celui de Boehm[46] (pour du C pur) possède une fonction comme `GC_malloc_atomic()`—en l'utilisant, vous déclarez que le bloc alloué avec cette fonction ne contiendra jamais de pointeur vers un autre bloc. Ça peut être une chaîne de texte, ou un autre type de donnée. (En effet, `GC_strdup()` appelle `GC_malloc_atomic()`.) Le GC ne va pas le balayer.

3.24 Optimisations de boucle

3.24.1 Optimisation étrange de boucle

Ceci est une des fonctions memcpy() les plus simple jamais implémentée:

```
void memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
        size_t i;
        for (i=0; i<cnt; i++)
                dst[i]=src[i];
};
```

Au moins MSVC 6.0 de la fin des années 1990 jusqu'à MSVC 2013 peuvent produire du code vraiment étrange (ce listing est généré par MSVC 2013 x86) :

```
_dst$ = 8               ; taille = 4
_src$ = 12              ; taille = 4
_cnt$ = 16              ; taille = 4
_memcpy PROC
        mov     edx, DWORD PTR _cnt$[esp-4]
        test    edx, edx
        je      SHORT $LN1@f
        mov     eax, DWORD PTR _dst$[esp-4]
        push    esi
        mov     esi, DWORD PTR _src$[esp]
        sub     esi, eax
; ESI=src-dst, i.e., différence des pointeurs
$LL8@f:
        mov     cl, BYTE PTR [esi+eax] ; charger l'octet em "esi+dst" ou en
    "src-dst+dst" au début ou juste en "src"
        lea     eax, DWORD PTR [eax+1] ; dst++
        mov     BYTE PTR [eax-1], cl   ; stocker l'octet en "(dst++)--" ou
    juste en "dst" au début
        dec     edx                    ; décrémenter le compteur jusqu'à ce
    que nous ayons fini
        jne     SHORT $LL8@f
        pop     esi
$LN1@f:
        ret     0
```

[46] https://www.hboehm.info/gc/

```
_memcpy ENDP
```

Ceci est étrange, car comment travaille les humains avec deux pointeurs? Ils stockent les deux adresses dans deux registres ou deux emplacements mémoire. Dans ce cas, le compilateur MSVC stocke les deux pointeurs comme un pointeur (*dst glissant* dans EAX) et la différence entre les pointeurs *src* et *dst* (qui reste inchangée lors de l'exécution du corps de la boucle) dans ESI. (À propos, ceci est un des rare cas où le type de donnée ptrdiff_t peut être utilisé.) Lorsqu'il doit charger un octet depuis *src*, il le charge en *diff* + *dst glissant* et stocke l'octet juste en *dst glissant*.

Ceci est plus une astuce d'optimisation. Mais j'ai récrit cette fonction en:

```
_f2     PROC
        mov     edx, DWORD PTR _cnt$[esp-4]
        test    edx, edx
        je      SHORT $LN1@f
        mov     eax, DWORD PTR _dst$[esp-4]
        push    esi
        mov     esi, DWORD PTR _src$[esp]
        ; eax=dst; esi=src
$LL8@f:
        mov     cl, BYTE PTR [esi+edx]
        mov     BYTE PTR [eax+edx], cl
        dec     edx
        jne     SHORT $LL8@f
        pop     esi
$LN1@f:
        ret     0
_f2     ENDP
```

…et ça fonctionne aussi efficacement que la version *optimisée* sur mon Intel Xeon E31220 @ 3.10GHz. Peut-être que cette optimisation ciblait des vieux CPUs x86 des années 1990, puisque ce truc est utilisé au moins par l'ancien MS VC 6.0?

Une idée?

Hex-Rays 2.2 a du mal à reconnaître des schémas comme ça (avec de la chance, temporairement?) :

```
void __cdecl f1(char *dst, char *src, size_t size)
{
  size_t counter; // edx@1
  char *sliding_dst; // eax@2
  char tmp; // cl@3

  counter = size;
  if ( size )
  {
    sliding_dst = dst;
    do
    {
      tmp = (sliding_dst++)[src - dst];         // difference (src-dst) is
    calculated once, at the beginnning
      *(sliding_dst - 1) = tmp;
```

```
      --counter;
    }
    while ( counter );
  }
}
```

Néanmoins, cette astuce d'optimisation est souvent utilisée par MSVC (pas uniquement dans des routines *memcpy()* DIY[47] maison, mais dans de nombreuses boucles qui utilisent deux tableaux ou plus). Donc, ça vaut le coup pour les rétro-ingénieurs de la garder à l'esprit.

3.24.2 Autre optimisation de boucle

Si vous traitez tous les éléments d'un tableau qui est situé dans la mémoire globale, le compilateur peut l'optimiser. Par exemple, calculons la somme de tous les éléments du tableau de 128 *int* :

```
#include <stdio.h>

int a[128];

int sum_of_a()
{
        int rt=0;

        for (int i=0; i<128; i++)
                rt=rt+a[i];

        return rt;
};

int main()
{
        // initialize
        for (int i=0; i<128; i++)
                a[i]=i;

        // calculate the sum
        printf ("%d\n", sum_of_a());
};
```

GCC 5.3.1 (x86) avec optimisation peut produire ceci (IDA) :

```
.text:080484B0 sum_of_a        proc near
.text:080484B0                 mov     edx, offset a
.text:080484B5                 xor     eax, eax
.text:080484B7                 mov     esi, esi
.text:080484B9                 lea     edi, [edi+0]
.text:080484C0
.text:080484C0 loc_80484C0:            ; CODE XREF: sum_of_a+1B
.text:080484C0                 add     eax, [edx]
```

[47] Do It Yourself

```
.text:080484C2                 add     edx, 4
.text:080484C5                 cmp     edx, offset ⤦
    ↳ __libc_start_main@@GLIBC_2_0
.text:080484CB                 jnz     short loc_80484C0
.text:080484CD                 rep retn
.text:080484CD sum_of_a        endp
.text:080484CD

...

.bss:0804A040                 public a
.bss:0804A040 a               dd 80h dup(?) ; DATA XREF: main:loc_8048338
.bss:0804A040                               ; main+19
.bss:0804A040 _bss            ends
.bss:0804A040
extern:0804A240 ; ===========================================================================
extern:0804A240
extern:0804A240 ; Segment type: Externs
extern:0804A240 ; extern
extern:0804A240                 extrn __libc_start_main@@GLIBC_2_0:near
extern:0804A240                             ; DATA XREF: main+25
extern:0804A240                             ; main+5D
extern:0804A244                 extrn __printf_chk@@GLIBC_2_3_4:near
extern:0804A248                 extrn __libc_start_main:near
extern:0804A248                             ; CODE XREF: ___libc_start_main
extern:0804A248                             ; DATA XREF: .got.plt:off_804A00C
```

Qu'est-ce que c'est que `__libc_start_main@@GLIBC_2_0` en `0x080484C5` ? Ceci est un label situé juste après la fin du tableau `a[]`. Cette fonction peut être récrite comme ceci:

```
int sum_of_a_v2()
{
        int *tmp=a;
        int rt=0;

        do
        {
                rt=rt+(*tmp);
                tmp++;
        }
        while (tmp<(a+128));

        return rt;
};
```

La première version a le compteur *i*, et l'adresse de chaque élément du tableau est calculée à chaque itération. La seconde version est plus optimisée: le pointeur sur chaque élément du tableau est toujours prêt et est déplacé de 4 octets à chaque itération. Comment vérifier si la boucle est terminée? Il suffit de comparer le pointeur avec l'adresse juste après la fin du tableau, qui est dans notre cas l'adresse de la fonction `__libc_start_main()` importée de la Glibc 2.0. Parfois ce genre de code

est perturbant, et ceci est une astuce d'optimisation très répandue, c'est pourquoi j'ai mis cet exemple.

Ma seconde version est très proche de ce que fait GCC, et lorsque je la compile, le code est presque le même que dans la première version, mais les deux première instructions sont échangées:

```
.text:080484D0                  public sum_of_a_v2
.text:080484D0 sum_of_a_v2      proc near
.text:080484D0                  xor     eax, eax
.text:080484D2                  mov     edx, offset a
.text:080484D7                  mov     esi, esi
.text:080484D9                  lea     edi, [edi+0]
.text:080484E0
.text:080484E0 loc_80484E0:             ; CODE XREF: sum_of_a_v2+1B
.text:080484E0                  add     eax, [edx]
.text:080484E2                  add     edx, 4
.text:080484E5                  cmp     edx, offset ⤦
    ↳ __libc_start_main@@GLIBC_2_0
.text:080484EB                  jnz     short loc_80484E0
.text:080484ED                  rep retn
.text:080484ED sum_of_a_v2      endp
```

Inutile de dire que cette optimisation n'est possible que si le compilateur peut calculer l'adresse de la fin du tableau pendant la compilation. Ceci se produit si le tableau est global et sa taille fixée.

Toutefois, si l'adresse du tableau est inconnue lors de la compilation, mais que la taille est fixée, l'adresse du label juste après la fin du tableau peut être calculée.

3.25 Plus sur les structures

3.25.1 Parfois une structure C peut être utilisée au lieu d'un tableau

Moyenne arithmétique

```
#include <stdio.h>

int mean(int *a, int len)
{
        int sum=0;
        for (int i=0; i<len; i++)
                sum=sum+a[i];
        return sum/len;
};

struct five_ints
{
        int a0;
        int a1;
        int a2;
```

```
        int a3;
        int a4;
};

int main()
{
        struct five_ints a;
        a.a0=123;
        a.a1=456;
        a.a2=789;
        a.a3=10;
        a.a4=100;
        printf ("%d\n", mean(&a, 5));
        // test:
    https://www.wolframalpha.com/input/?i=mean(123,456,789,10,100)
};
```

Ceci fonctionne: la fonction *mean()* ne va jamais accéder après la fin de la structure *five_ints*, car 5 est passé, signifiant que seuls 5 entiers vont être accédés.

Mettre une chaîne dans une structure

```
#include <stdio.h>

struct five_chars
{
        char a0;
        char a1;
        char a2;
        char a3;
        char a4;
} __attribute__ ((aligned (1),packed));

int main()
{
        struct five_chars a;
        a.a0='h';
        a.a1='i';
        a.a2='!';
        a.a3='\n';
        a.a4=0;
        printf (&a); // prints "hi!"
};
```

L'attribut *((aligned (1),packed))* doit être utilisé, car sinon, chaque champ de la structure sera aligné sur une limite de 4 ou 8 octets.

Résumé

Ceci est simplement un autre exemple de la façon dont les structures et les tableaux sont stockés en mémoire. Peut-être qu'aucun programmeur sain ne ferait quelque

chose comme dans cet exemple, excepté dans le cas d'astuces spécifiques. Ou peut-être dans le cas d'obfuscation de code source?

3.25.2 Tableau non dimensionné dans une structure C

Nous pouvons trouver certaines structures win32 avec la dernier champ défini comme un tableau d'un élément.

```
typedef struct _SYMBOL_INFO {
  ULONG   SizeOfStruct;
  ULONG   TypeIndex;

  ...

  ULONG   MaxNameLen;
  TCHAR   Name[1];
} SYMBOL_INFO, *PSYMBOL_INFO;
```

(https://msdn.microsoft.com/en-us/library/windows/desktop/ms680686(v=vs.85).aspx)

Ceci est une astuce, signifiant que le dernier champ est un tableau de taille inconnue, qui doit être calculée lors de l'allocation de la structure.

Pourquoi: le champ *Name* peut-être court, donc pourquoi le définir avec une sorte de constante *MAX_NAME* qui peut être 128, 256 et même plus ?

Pourquoi ne pas utiliser un pointeur à la place ? Alors vous devez allouer deux blocs: un pour la structure et pour la chaîne. Ceci peut-être plus lent et peut nécessiter un plus large surplus de mémoire. Donc, vous devez déréférencer le pointeur (i.e., lire l'adresse de la chaîne dans la structure)—ce n'est pas un problème, mais certains disent que c'est un coût supplémentaire.

Ceci est connu comme l'*astuce struct* : http://c-faq.com/struct/structhack.html.

Exemple:

```
#include <stdio.h>

struct st
{
        int a;
        int b;
        char s[];
};

void f (struct st *s)
{
        printf ("%d %d %s\n", s->a, s->b, s->s);
        // f() can't replace s[] with bigger string - size of allocated block
    is unknown at this point
};
```

```
int main()
{
#define STRING "Hello!"
        struct st *s=malloc(sizeof(struct st)+strlen(STRING)+1); // incl.
    terminating zero
        s->a=1;
        s->b=2;
        strcpy (s->s, STRING);
        f(s);
};
```

En quelques mots, ça fonctionne car le C n'a pas de vérification des bornes d'un tableau. Tout tableau est traité comme ayant une taille infinie.

Problème: après l'allocation, la taille entière du bloc alloué pour la structure est inconnue (excepté pour le gestionnaire de mémoire), donc vous ne pouvez pas remplacer une chaîne par un chaîne plus large. Vous pourriez faire quelque chose comme ça si le champ était déclaré comme quelque chose comme *s[MAX_NAME]*.

Autrement dit, vous avez une structure plus un tableau (ou une chaîne) fusionnés ensemble dans un bloc de mémoire alloué unique. Un autre problème est que vous ne pouvez évidemment pas déclarer deux tableaux comme ceci dans une structure unique, ou déclarer un autre champ après un tel tableau.

Les cieux compilateurs nécessitent de déclarer le tableau avec au moins un élément: *s[1]*, les plus récents permettent de le déclarer comme un tableau de taille variable:*s[]*. Ceci est aussi appelé *membre tableau flexible*.

En lire plus à ce sujet dans GCC documentation[48], MSDN documentation[49].

Dennis Ritchie (un des créateurs du C) a appelé ce truc «amabilité non voulue avec l'implémentation du C» (peut-être pour reconnaître la nature astucieuse de cette ruse).

Aimez le ou non, utilisez le ou non: il est encore une autre démonstration de la façon dont les structures sont stockées dans la mémoire, c'est pourquoi j'en ai parlé.

3.25.3 Version de structure C

De nombreux programmeurs Windows ont vu ceci dans MSDN:

```
SizeOfStruct
    The size of the structure, in bytes. This member must be set to sizeof(↙
    ↳ SYMBOL_INFO).
```

(https://msdn.microsoft.com/en-us/library/windows/desktop/ms680686(v=vs.85).aspx)

En effet, certaines structures comme *SYMBOL_INFO* commencent en effet avec ce champ. Pourquoi ? C'est une sorte de version de structure.

[48] https://gcc.gnu.org/onlinedocs/gcc/Zero-Length.html
[49] https://msdn.microsoft.com/en-us/library/b6fae073.aspx

Imaginez que vous avez une fonction qui dessine un cercle. Elle prend un unique argument—un pointeur sur une structure avec seulement trois champs: X, Y et radius. Et puis, les affichages couleurs ont inondé le marché durant les années 1980. Et vous voulez ajouter un argument *color* à la fonction. Mais, disons que vous ne pouvez pas lui ajouter un argument (de nombreux logiciels utilisent votre API[50] et ne peuvent pas être recompilés). Et si un vieux logiciels utilise votre API avec un affichage couleur, faites que votre fonction dessine un cercle avec par défaut les couleurs noire et blanche.

Un autre jour, vous ajoutez une autre possibilité: le cercle peut maintenant être rempli, et le type de brosse peut être passé.

Voici une solution à ce problème:

```
#include <stdio.h>

struct ver1
{
        size_t SizeOfStruct;
        int coord_X;
        int coord_Y;
        int radius;
};

struct ver2
{
        size_t SizeOfStruct;
        int coord_X;
        int coord_Y;
        int radius;
        int color;
};

struct ver3
{
        size_t SizeOfStruct;
        int coord_X;
        int coord_Y;
        int radius;
        int color;
        int fill_brush_type; // 0 - do not fill circle
};

void draw_circle(struct ver3 *s) // latest struct version is used here
{
        //
    we presume SizeOfStruct, coord_X and coord_Y fields are always present
        printf ("We are going to draw a circle at %d:%d\n", s->coord_X, s->↙
    ↘ coord_Y);

        if (s->SizeOfStruct>=sizeof(int)*5)
        {
```

[50] Application Programming Interface

```
                // this is at least ver2, color field is present
                printf ("We are going to set color %d\n", s->color);
        }

        if (s->SizeOfStruct>=sizeof(int)*6)
        {
                // this is at least ver3, fill_brush_type field is present
                printf ("We are going to fill it using brush type %d\n", s↙
    ↪ ->fill_brush_type);
        }
};

// early software version
void call_as_ver1()
{
        struct ver1 s;
        s.SizeOfStruct=sizeof(s);
        s.coord_X=123;
        s.coord_Y=456;
        s.radius=10;
        printf ("** %s()\n", __FUNCTION__);
        draw_circle(&s);
};

// next software version
void call_as_ver2()
{
        struct ver2 s;
        s.SizeOfStruct=sizeof(s);
        s.coord_X=123;
        s.coord_Y=456;
        s.radius=10;
        s.color=1;
        printf ("** %s()\n", __FUNCTION__);
        draw_circle(&s);
};

// latest, the most extended version
void call_as_ver3()
{
        struct ver3 s;
        s.SizeOfStruct=sizeof(s);
        s.coord_X=123;
        s.coord_Y=456;
        s.radius=10;
        s.color=1;
        s.fill_brush_type=3;
        printf ("** %s()\n", __FUNCTION__);
        draw_circle(&s);
};

int main()
{
```

```
        call_as_ver1();
        call_as_ver2();
        call_as_ver3();
};
```

Autrement dit, le champ *SizeOfStruct* prend le rôle d'un champ *version of structure*. Il pourrait être un type énuméré (1, 2, 3, etc.), mais mettre le champ *SizeOfStruct* à *sizeof(struct...)* est moins sujet à l'erreur: nous écrivons simplement *s.SizeOfStruct=sizeof(...)* dans le code de l'appelant.

En C++, ce problème est résolu en utilisant l'*hérutage* (3.21.1 on page 706). Vous avez seulement à étendre la classe de base (appelons la *Circle*), puis vous aurez une classe *ColoredCircle*, et ensuite *FilledColoredCircle*, et ainsi de suite. La version courante d'un objet (ou, plus précisément, le *type* courant) sera déterminé en utilisant la RTTI de C++.

Donc lorsque vous voyez *SizeOfStruct* quelque part dans MSDN—peut-être que cette structure a été étendue au moins une fois par le passé.

3.25.4 Fichier des meilleurs scores dans le jeu «Block out» et sérialisation basique

De nombreux jeux vidéo ont un fichier des meilleurs scores, parfois appelé «Hall of fame». L'ancien jeu «Block out»[51] (tetris 3D de 1989) ne fait pas exception, voici ce que nous voyons à la fin:

```
 ***    H A L L    O F    F A M E    ***

   Pit: 7x7x18,   Block Set: FLAT

  1. Xenia.....       479 (03-27-2018)
  2. Paul......       353 (03-27-2018)
  3. John......       326 (03-27-2018)
  4. James.....       324 (03-27-2018)
  5. Charlie...       234 (03-27-2018)
  6. Mike......       181 (03-27-2018)
  7. Phil......       172 (03-27-2018)
  8. Mary......       123 (03-27-2018)
  9. Tom.......       119 (03-27-2018)
 10. Bob.......       119 (03-27-2018)

              Start Game
              Change Setup
              Main Menu
              Quit Game
```

Fig. 3.4: Table des meilleurs scores

Maintenant, nous pouvons voir que le fichier qui a changé après que nous ayons ajouté notre nom est *BLSCORE.DAT*.

[51] http://www.bestoldgames.net/eng/old-games/blockout.php

```
% xxd -g 1 BLSCORE.DAT

00000000: 0a 00 58 65 6e 69 61 2e 2e 2e 2e 2e 00 df 01 00  ..Xenia.........
00000010: 00 30 33 2d 32 37 2d 32 30 31 38 00 50 61 75 6c  .03-27-2018.Paul
00000020: 2e 2e 2e 2e 2e 2e 00 61 01 00 00 30 33 2d 32 37  .......a...03-27
00000030: 2d 32 30 31 38 00 4a 6f 68 6e 2e 2e 2e 2e 2e 2e  -2018.John......
00000040: 00 46 01 00 00 30 33 2d 32 37 2d 32 30 31 38 00  .F...03-27-2018.
00000050: 4a 61 6d 65 73 2e 2e 2e 2e 2e 00 44 01 00 00 30  James......D...0
00000060: 33 2d 32 37 2d 32 30 31 38 00 43 68 61 72 6c 69  3-27-2018.Charli
00000070: 65 2e 2e 2e 00 ea 00 00 00 30 33 2d 32 37 2d 32  e........03-27-2
00000080: 30 31 38 00 4d 69 6b 65 2e 2e 2e 2e 2e 2e 00 b5  018.Mike........
00000090: 00 00 00 30 33 2d 32 37 2d 32 30 31 38 00 50 68  ...03-27-2018.Ph
000000a0: 69 6c 2e 2e 2e 2e 2e 2e 00 ac 00 00 00 30 33 2d  il...........03-
000000b0: 32 37 2d 32 30 31 38 00 4d 61 72 79 2e 2e 2e 2e  27-2018.Mary....
000000c0: 2e 2e 00 7b 00 00 00 30 33 2d 32 37 2d 32 30 31  ...{...03-27-201
000000d0: 38 00 54 6f 6d 2e 2e 2e 2e 2e 2e 2e 00 77 00 00  8.Tom........w..
000000e0: 00 30 33 2d 32 37 2d 32 30 31 38 00 42 6f 62 2e  .03-27-2018.Bob.
000000f0: 2e 2e 2e 2e 2e 2e 00 77 00 00 00 30 33 2d 32 37  .......w...03-27
00000100: 2d 32 30 31 38 00                                -2018.
```

Toutes les entrées sont clairement visibles. Le premier octet est probablement le nombre d'entrées. Le second est zéro, en fait, le nombre d'entrées peut-être une valeurs 16-bit couvrant les deux premiers octets.

Ensuite, après le nom «Xenia», nous voyons les octets 0xDF et 0x01. Xenia a un score de 479, et ceci est exactement 0x1DF en hexadécimal. Donc une valeur de score est probablement un entier 16-bit, ou un entier 32-bit: il y a deux octets à zéro de plus après.

Maintenant, pensons au fait qu'à la fois les éléments des tableaux et des structures sont toujours placés en mémoire de manière adjacente les uns aux autres. Cela nous permet d'écrire le tableau/la structure entièrement dans le fichier en utilisant une fonction unique *write()* ou *fwrite()*, et de le restaurer en utilisant *read()* ou *fread()*, aussi simplement que ça. Ceci est ce qui est appelé *sérialisation* de nos jours.

Lire

Maintenant, écrivons un petit programme en C pour lire le fichier des meilleurs scores:

```
#include <assert.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>

struct entry
{
        char name[11]; // incl. terminating zero
        uint32_t score;
        char date[11]; // incl. terminating zero
} __attribute__ ((aligned (1),packed));
```

```
struct highscore_file
{
        uint8_t count;
        uint8_t unknown;
        struct entry entries[10];
} __attribute__ ((aligned (1), packed));

struct highscore_file file;

int main(int argc, char* argv[])
{
        FILE* f=fopen(argv[1], "rb");
        assert (f!=NULL);
        size_t got=fread(&file, 1, sizeof(struct highscore_file), f);
        assert (got==sizeof(struct highscore_file));
        fclose(f);
        for (int i=0; i<file.count; i++)
        {
                printf ("name=%s score=%d date=%s\n",
                                file.entries[i].name,
                                file.entries[i].score,
                                file.entries[i].date);
        };
};
```

Nous avons besoin de l'attribut *((aligned (1),packed))* de GCC afin que tous les champs de la structure soient alignés sur une limite de 1-octet.

Bien sûr, il fonctionne:

```
name=Xenia..... score=479 date=03-27-2018
name=Paul...... score=353 date=03-27-2018
name=John...... score=326 date=03-27-2018
name=James..... score=324 date=03-27-2018
name=Charlie... score=234 date=03-27-2018
name=Mike...... score=181 date=03-27-2018
name=Phil...... score=172 date=03-27-2018
name=Mary...... score=123 date=03-27-2018
name=Tom....... score=119 date=03-27-2018
name=Bob....... score=119 date=03-27-2018
```

(Inutile de dire que chaque nom est complété avec des points, à la fois à l'écran et dans le fichier, peut-être pour des raisons esthétique.)

Écrire

Vérifions si nous avons raison à propos de la largeur de la variable du score. Est-ce réellement sur 32 bits?

```
int main(int argc, char* argv[])
{
        FILE* f=fopen(argv[1], "rb");
        assert (f!=NULL);
```

```
        size_t got=fread(&file, 1, sizeof(struct highscore_file), f);
        assert (got==sizeof(struct highscore_file));
        fclose(f);

        strcpy (file.entries[1].name, "Mallory...");
        file.entries[1].score=12345678;
        strcpy (file.entries[1].date, "08-12-2016");

        f=fopen(argv[1], "wb");
        assert (f!=NULL);
        got=fwrite(&file, 1, sizeof(struct highscore_file), f);
        assert (got==sizeof(struct highscore_file));
        fclose(f);
};
```

Lançons Blockout:

```
***   H A L L   O F   F A M E   ***

 Pit: 7x7x18,  Block Set: FLAT

  1. Xenia.....       479 (03-27-2018)
  2. Mallory...    345678 (08-12-2016)
  3. John......       326 (03-27-2018)
  4. James.....       324 (03-27-2018)
  5. Charlie...       234 (03-27-2018)
  6. Mike......       181 (03-27-2018)
  7. Phil......       172 (03-27-2018)
  8. Mary......       123 (03-27-2018)
  9. _                123 (03-27-2018)
 10. Tom.......       119 (03-27-2018)
```

Fig. 3.5: Table des meilleurs scores

Les deux premiers chiffres (1 et 2) ne sont pas affichés: 12345678 devient 345678. Peut-être est-ce un problème de formatage... mais le nombre est presque correct. Maintenant, je le change en 999999 et relance le jeu:

```
***    H A L L    O F    F A M E    ***

  Pit: 7x7x18,   Block Set: FLAT

 1.  Xenia.....        479  (03-27-2018)
 2.  Mallory...     999999  (08-12-2016)
 3.  John......        326  (03-27-2018)
 4.  James.....        324  (03-27-2018)
 5.  Charlie...        234  (03-27-2018)
 6.  Mike......        181  (03-27-2018)
 7.  Phil......        172  (03-27-2018)
 8.                    133  (03-27-2018)
 9.  ..........        132  (03-27-2018)
10.  Mary......        123  (03-27-2018)
```

Fig. 3.6: Table des meilleurs scores

Maintenant, c'est correct. Oui, la valeur du score est un entier 32-bit.

Est-ce de la sérialisation?

…presque. Ce genre de sérialisation est très populaire dans les logiciels scientifiques et d'ingénierie, où l'efficacité et la rapidité sont bien plus importantes que de convertir de et vers XML[52] ou JSON[53].

Une chose importante est que vous ne pouvez évidemment pas sérialiser des pointeurs, car à chaque fois que vous chargez le programme en mémoire, toutes les structures peuvent être allouées à des endroits différents.

Mais, si vous travaillez sur des sortes de MCU à bas coût avec un simple OS dessus et que vous avez vos structures toujours allouées à la même place en mémoire, peut-être pouvez-vous sauver et restaurer de la sorte.

Bruit aléatoire

Lorsque je préparais cet exemple, j'ai dû lancer «Block out» de nombreuses fois et jouer un peu avec pour remplir la table des meilleurs scores avec des noms au hasard.

Et lorsqu'il y avait seulement 3 entrées dans le fichier, j'ai vu ceci:

```
00000000: 03 00 54 6f 6d 61 73 2e 2e 2e 2e 2e 00 da 2a 00  ..Tomas.......*.
00000010: 00 30 38 2d 31 32 2d 32 30 31 36 00 43 68 61 72  .08-12-2016.Char
00000020: 6c 69 65 2e 2e 2e 00 8b 1e 00 00 30 38 2d 31 32  lie........08-12
00000030: 2d 32 30 31 36 00 4a 6f 68 6e 2e 2e 2e 2e 2e 2e  -2016.John......
00000040: 00 80 00 00 00 30 38 2d 31 32 2d 32 30 31 36 00  .....08-12-2016.
00000050: 00 00 57 c8 a2 01 06 01 ba f9 47 c7 05 00 f8 4f  ..W.......G....O
00000060: 06 01 06 01 a6 32 00 00 00 00 00 00 00 00 00 00  .....2..........
```

[52] Extensible Markup Language
[53] JavaScript Object Notation

```
00000070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000a0: 00 00 00 00 00 00 00 00 00 00 93 c6 a2 01 46 72  ..............Fr
000000b0: 8c f9 f6 c5 05 00 f8 4f 00 02 06 01 a6 32 06 01  .......O.....2..
000000c0: 00 00 98 f9 f2 c0 05 00 f8 4f 00 02 a6 32 a2 f9  .........O...2..
000000d0: 80 c1 a6 32 a6 32 f4 4f aa f9 39 c1 a6 32 06 01  ...2.2.O..9..2..
000000e0: b4 f9 2b c5 a6 32 e1 4f c7 c8 a2 01 82 72 c6 f9  ..+..2.O.....r..
000000f0: 30 c0 05 00 00 00 00 00 00 00 a6 32 d4 f9 76 2d  0.........2..v-
00000100: a6 32 00 00 00 00                                .2....
```

Le premier octet a la valeur 3, signifiant qu'il y a 3 entrées. Et ces 3 entrées sont présentes. Mais nous avons des valeurs aléatoires dans la seconde moitié du fichier.

Le bruit provient probablement de données non initialisées. Peut-être que «Block out» alloue de la mémoire pour 10 entrées quelque part dans le tas, où, manifestement, des valeurs pseudo-aléatoires (laissées par quelque chose d'autre) sont présentes. Ensuite il a rempli les premier/second octet, 3 entrées, et puis n'a jamais touché aux 7 autres entrées, donc elles ont été écrites dans le fichier telles quelles.

Lorsque «Block out» charge le fichier des meilleurs scores la fois suivante, il lit le nombre d'entrée dans les 2 premiers octets (3) et puis ignore ce qui vient après elles.

Ceci est un problème courant. Pas un problème au sens strict: ce n'est pas un bogue, mais de l'information peut fuiter à l'extérieur.

Les versions de Microsoft Word des années 1990 laissaient souvent des morceaux de texte précédemment édité dans les fichiers *.doc*. C'était alors une sorte de distraction d'obtenir un fichier *.doc* de quelqu'un d'autre, de l'ouvrir dans un éditeur hexadécimal et de lire d'autres choses, qui avaient été éditées avant sur cet ordinateur.

Le problème peut être beaucoup plus sérieux: le bogue Heartbleed dans OpenSSL.

Devoir

«Block out» a plusieurs types de pièces (plat/basique/étendu), la taille peut être configurée, etc. Et il semble que pour chaque configuration, «Block out» a son propre tableau des meilleurs scores. J'ai remarqué que de l'information est probablement stockée dans le fichier *BLSCORE.IDX*. Ceci peut être un travail pour les fans de «Block out»—de comprendre aussi sa structure. Les fichiers de «Block out» sont ici: `http://beginners.re/examples/blockout.zip` (incluant le fichier binaire des meilleurs scores que j'ai utilisé dans cet exemple). Vous pouvez utiliser DosBox pour le lancer.

3.26 memmove() et memcpy()

La différence entre ces deux fonctions standards est que *memcpy()* copie aveuglément un bloc à un autre endroit, alors que *memmove()* gère correctement les blocs qui se recouvrent. Par exemple, si vous voulez déplacer une chaîne de deux octets en arrière:

```
`|.|.|h|e|l|l|o|...` -> `|h|e|l|l|o|...`
```

memcpy() qui copie des mots de 32-bit ou de 64-bit à la fois, ou même SIMD, va manifestement échouer ici, une routine de copie par octet doit être utilisée à la place.

Maintenant un exemple encore plus avancé, insérer deux octets au début d'une chaîne:

```
`|h|e|l|l|o|...` -> `|.|.|h|e|l|l|o|...`
```

Maintenant, même une copie octet par octet va échouer, car vous devez copier en partant de la fin.

C'est un cas rare où le flag x86 DF doit être mis avant l'instruction REP MOVSB : DF défini la direction, et maintenant, nous devons déplacer en arrière.

La routine *memmove()* typique fonctionne comme ceci: 1) si la source est avant la destination, copier en avant; 2) si la source est après la destination, copier en arrière.

Ceci est la fonction *memmove()* de uClibc:

```
void *memmove(void *dest, const void *src, size_t n)
{
	int eax, ecx, esi, edi;
	__asm__ __volatile__(
		"	movl	%%eax, %%edi\n"
		"	cmpl	%%esi, %%eax\n"
		"	je	2f\n" /* (optional) src == dest -> NOP */
		"	jb	1f\n" /* src > dest -> simple copy */
		"	leal	-1(%%esi,%%ecx), %%esi\n"
		"	leal	-1(%%eax,%%ecx), %%edi\n"
		"	std\n"
		"1:	rep; movsb\n"
		"	cld\n"
		"2:\n"
		: "=&c" (ecx), "=&S" (esi), "=&a" (eax), "=&D" (edi)
		: "0" (n), "1" (src), "2" (dest)
		: "memory"
	);
	return (void*)eax;
}
```

Dans le premier cas, REP MOVSB est appelée avec le flag DF à zéro. Dans le second, DF est mis, puis remis à zéro.

Un algorithme plus complexe contient la logique suivante:

«Si la différence entre la *source* et la *destination* est plus grande que la largeur d'un mot, copier en utilisant des mots plutôt que des octets, et utiliser une copie octet par octet pour copier les parties non alignées.»

Voici comment ça se passe dans la partie C non optimisée de la Glibc 2.24.

Compte tenu de cela, *memmove()* peut être plus lente que *memcpy()*. Mais certains, Linus Torvalds inclus, argumentent[54] que *memcpy()* devrait être un alias (ou synonyme) de *memmove()*, et cette dernière fonction devrait juste tester au début, si les buffers se recouvrent ou non, et ensuite se comporter comme *memcpy()* ou *memmove()*. De nos jours, le test de recouvrement de buffers est peu coûteux, après tout.

3.26.1 Stratagème anti-debugging

J'ai entendu parler de stratagème anti-debugging où tout ce que vous devez faire pour crasher le processus est de mettre DF : le prochain appel à *memcpy()* va conduire au crash, car il copiera en arrière. Mais je ne peux pas tester ceci: il semble que toutes les routines de copie de mémoire mettent DF à 1/0 comme elles le veulent. D'un autre côté, *memmove()* de uClibc que j'ai déjà cité ici, ne remet pas explicitement DF à zéro (assume-t-elle que DF est toujours à zéro?), donc ça peut vraiment planter.

3.27 setjmp/longjmp

Il s'agit d'un mécanisme en C qui est très similaire au mecanisme throw/catch en C++ ou d'autres LPs de haut niveau. Voici un exemple tiré de la zlib:

```
...

    /* return if bits() or decode() tries to read past available input */
    if (setjmp(s.env) != 0)             /* if came back here via longjmp(),↙
    ↳  */
        err = 2;                        /*  then skip decomp(), return ↙
    ↳ error */
    else
        err = decomp(&s); /* decompress */

...

    /* load at least need bits into val */
    val = s->bitbuf;
    while (s->bitcnt < need) {
        if (s->left == 0) {
            s->left = s->infun(s->inhow, &(s->in));
            if (s->left == 0) longjmp(s->env, 1); /* out of input */

...

        if (s->left == 0) {
            s->left = s->infun(s->inhow, &(s->in));
            if (s->left == 0) longjmp(s->env, 1); /* out of input */
```

(zlib/contrib/blast/blast.c)

[54] https://bugzilla.redhat.com/show_bug.cgi?id=638477#c132

L'appel à `setjmp()` sauve les valeurs courantes de PC, SP et autres registres dans une structure env, puis renvoie 0.

En cas d'erreur, `longjmp()` vous *téléporte* au point juste après l'appel à `setjmp()`, comme si l'appel à `setjmp()` avait renvoyé une valeur non nulle (qui avait été passée à `longjmp()`). Ceci nous rappel l'appel système `fork()` sous UNIX.

Maintenant, regardons un exemple épuré:

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf env;

void f2()
{
        printf ("%s() begin\n", __FUNCTION__);
        // something odd happened here
        longjmp (env, 1234);
        printf ("%s() end\n", __FUNCTION__);
};

void f1()
{
        printf ("%s() begin\n", __FUNCTION__);
        f2();
        printf ("%s() end\n", __FUNCTION__);
};

int main()
{
        int err=setjmp(env);
        if (err==0)
        {
                f1();
        }
        else
        {
                printf ("Error %d\n", err);
        };
};
```

Si nous le lançons, nous voyons:

```
f1() begin
f2() begin
Error 1234
```

La structure `jmp_buf` est généralement non-documentée, pour préserver la compatibilité ascendante.

Regardons comment setjmp() est implémenté dans MSVC 2013 x64:

```
                ...
```

```
; RCX = address of jmp_buf

mov     [rcx], rax
mov     [rcx+8], rbx
mov     [rcx+18h], rbp
mov     [rcx+20h], rsi
mov     [rcx+28h], rdi
mov     [rcx+30h], r12
mov     [rcx+38h], r13
mov     [rcx+40h], r14
mov     [rcx+48h], r15
lea     r8, [rsp+arg_0]
mov     [rcx+10h], r8
mov     r8, [rsp+0]      ; get saved RA from stack
mov     [rcx+50h], r8    ; save it
stmxcsr dword ptr [rcx+58h]
fnstcw  word ptr [rcx+5Ch]
movdqa  xmmword ptr [rcx+60h], xmm6
movdqa  xmmword ptr [rcx+70h], xmm7
movdqa  xmmword ptr [rcx+80h], xmm8
movdqa  xmmword ptr [rcx+90h], xmm9
movdqa  xmmword ptr [rcx+0A0h], xmm10
movdqa  xmmword ptr [rcx+0B0h], xmm11
movdqa  xmmword ptr [rcx+0C0h], xmm12
movdqa  xmmword ptr [rcx+0D0h], xmm13
movdqa  xmmword ptr [rcx+0E0h], xmm14
movdqa  xmmword ptr [rcx+0F0h], xmm15
retn
```

Cela remplit juste la structure jmp_buf avec la valeur courante de presque tous les registres. Aussi, la valeur courante de RA est prise de la pile et sauvée dans jmp_buf: elle sera utilisée comme nouvelle valeur de PC dans le futur.

Maintenant longjmp() :

```
...

; RCX = address of jmp_buf

mov     rax, rdx
mov     rbx, [rcx+8]
mov     rsi, [rcx+20h]
mov     rdi, [rcx+28h]
mov     r12, [rcx+30h]
mov     r13, [rcx+38h]
mov     r14, [rcx+40h]
mov     r15, [rcx+48h]
ldmxcsr dword ptr [rcx+58h]
fnclex
fldcw   word ptr [rcx+5Ch]
movdqa  xmm6, xmmword ptr [rcx+60h]
movdqa  xmm7, xmmword ptr [rcx+70h]
```

```
                movdqa  xmm8, xmmword ptr [rcx+80h]
                movdqa  xmm9, xmmword ptr [rcx+90h]
                movdqa  xmm10, xmmword ptr [rcx+0A0h]
                movdqa  xmm11, xmmword ptr [rcx+0B0h]
                movdqa  xmm12, xmmword ptr [rcx+0C0h]
                movdqa  xmm13, xmmword ptr [rcx+0D0h]
                movdqa  xmm14, xmmword ptr [rcx+0E0h]
                movdqa  xmm15, xmmword ptr [rcx+0F0h]
                mov     rdx, [rcx+50h]  ; get PC (RIP)
                mov     rbp, [rcx+18h]
                mov     rsp, [rcx+10h]
                jmp     rdx             ; jump to saved PC

                ...
```

Cela restaure (presque) tous les registres, prend RA dans la structure et y saute. Ceci fonctionne en effet comme si setjmp() retournait à l'appelant. Aussi, RAX est mis pour être égal au second argument de longjmp(). Ceci fonctionne comme si setjmp() renvoyait une valeur non-zéro en première place.

Comme effet de bord de la restauration de SP, toutes les valeurs dans la pile qui ont été définies et utilisées entre les appels à setjmp() et longjmp() sont laissées tomber. Elles ne seront plus utilisées du tout. Ainsi, longjmp() saute usuellement en arrière [55].

Ceci implique que, contrairement au mécanisme throw/catch en C++, aucune mémoire ne sera libérée, aucun destructeur ne sera appelé, etc. Ainsi, cette technique peut parfois être dangereuse. Néanmoins, elle est assez populaire. C'est toujours utilisé dans Oracle RDBMS.

Cela a aussi un effet de bord inattendu: si un buffer a été dépassé dans une des fonctions (peut-être à cause d'une attaque distante), et qu'une fonction veut signaler une erreur, et ça appelle longjmp(), la partie de la pile récrite ne sera pas utilisée.

À titre d'exercice, vous pouvez essayer de comprendre pourquoi tous les registres ne sont pas sauvegardés. Pourquoi XMM0-XMM5 et d'autres registres sont évités?

3.28 Autres hacks bizarres de la pile

3.28.1 Accéder aux arguments/variables locales de l'appelant

Des bases de C/C++, nous savons qu'il est impossible à une fonction d'accéder aux arguments de la fonction appelante ou à ses variables locales.

Néanmoins, c'est possible en utilisant des astuces tordues. Par exemple:

```
#include <stdio.h>

void f(char *text)
```

[55] Toutefois, il y a des gens qui l'utilisent pour des choses bien plus compliquées, imitation des coroutines, etc.: https://www.embeddedrelated.com/showarticle/455.php, http://fanf.livejournal.com/105413.html

```
{
        // print stack
        int *tmp=&text;
        for (int i=0; i<20; i++)
        {
                printf ("0x%x\n", *tmp);
                tmp++;
        };
};

void draw_text(int X, int Y, char* text)
{
        f(text);

        printf ("We are going to draw [%s] at %d:%d\n", text, X, Y);
};

int main()
{
        printf ("address of main()=0x%x\n", &main);
        printf ("address of draw_text()=0x%x\n", &draw_text);
        draw_text(100, 200, "Hello!");
};
```

Sur Ubuntu 32-bit avec GCC 5.4.0, j'obtiens ceci:

```
address of main()=0x80484f8
address of draw_text()=0x80484cb
0x8048645       first argument to f()
0x8048628
0xbfd8ab98
0xb7634590
0xb779eddc
0xb77e4918
0xbfd8aba8
0x8048547       return address into the middle of main()
0x64            first argument to draw_text()
0xc8            second argument to draw_text()
0x8048645       third argument to draw_text()
0x8048581
0xb779d3dc
0xbfd8abc0
0x0
0xb7603637
0xb779d000
0xb779d000
0x0
0xb7603637
```

(Les commentaires sont miens.)

Puisque *f()* commence à énumérer les éléments de la pile à son premier argument, le premier élément de la pile est en effet un pointeur sur la chaîne «Hello! ». Nous

voyons que son adresse est aussi utilisée comme troisième argument de la fonction *draw_text()*.

Dans *f()* nous pouvons lire tous les arguments des fonctions et les variables locales si nous connaissons exactement l'agencement de la pile, mais ça change toujours d'un compilateur à l'autre. Des niveaux d'optimisation différents modifient grandement la structure de la pile.

Mais, si nous pouvons d'une manière ou d'une autre détecter l'information dont nous avons besoin, nous pouvons l'utiliser et même la modifier. À titre d'exemple, j'ai retravaillé la fonction *f()* :

```
void f(char *text)
{
        ...

        // find 100, 200 values pair and modify the second on
        tmp=&text;
        for (int i=0; i<20; i++)
        {
                if (*tmp==100 && *(tmp+1)==200)
                {
                        printf ("found\n");
                        *(tmp+1)=210; // change 200 to 210
                        break;
                };
                tmp++;
        };
};
```

Hé mais, ça fonctionne:

```
found
We are going to draw [Hello!] at 100:210
```

Résumé

C'est vraiment un sale hack, dont le but est de montrer l'intérieur de la pile. Je n'ai jamais vu ni entendu dire que quelqu'un ai utilisé ceci dans du code réel. Mais encore, ceci est un bon exemple.

Exercice

L'exemple a été compilé sans optimisation sur Ubuntu 32-bit avec GCC 5.4.0 et il fonctionne. Mais lorsque j'active l'optimisation maximum -03, ça plante. Essayez de trouver pourquoi.

Utilisez votre compilateur et OS favori, essayez différents niveaux d'optimisation, trouvez si ça fonctionne et si ça ne fonctionne pas, trouvez pourquoi.

3.28.2 Renvoyer une chaîne

Ceci est un bug classique tiré de Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999) :

```
#include <stdio.h>

char* amsg(int n, char* s)
{
        char buf[100];

        sprintf (buf, "error %d: %s\n", n, s) ;

        return buf;
};

int main()
{
        printf ("%s\n", amsg (1234, "something wrong!"));
};
```

Il va planter. Tout d'abord essayons de comprendre pourquoi.

Ceci est l'état de la pile avant le retour de amsg() :

```
(lower addresses)

...

[amsg(): 100 bytes]
[RA]                    <- current SP
[two amsg arguments]
[something else]
[main() local variables]

...

(upper addresses)
```

Ensuite amsg() rend le contrôle du flux à `main()`, jusqu'ici, tout va bien. Mais `printf()` est appelée depuis `main()`, qui, en fait, utilise la pile pour ses propres besoin, zappant le buffer de 100-octet. Au mieux, du contenu indéterminé sera affiché.

Difficile à croire, mais je sais comment résoudre ce problème:

```
#include <stdio.h>

char* amsg(int n, char* s)
{
        char buf[100];

        sprintf (buf, "error %d: %s\n", n, s) ;

        return buf;
```

```
};

char* interim (int n, char* s)
{
        char large_buf[8000];
        // make use of local array.
        // it will be optimized away otherwise, as useless.
        large_buf[0]=0;
        return amsg (n, s);
};

int main()
{
        printf ("%s\n", interim (1234, "something wrong!"));
};
```

Cela va fonctionner si il est compilé avec MSVC 2013 sans optimisation et avec l'option `/GS-` option[56]. MSVC avertira: "warning C4172: returning address of local variable or temporary", mais le code s'exécutera et le message sera affiché. Regardons l'état de la pile au moment où amsg() renvoie le contrôle à interim() :

```
(lower addresses)

...

[amsg(): 100 bytes]
[RA]                                    <- current SP
[two amsg() arguments]
[interim() stuff, incl. 8000 bytes]
[something else]
[main() local variables]

...

(upper addresses)
```

Maintenant, l'état de la pile au moment où interim() rend le contrôle à `main()` :

```
(lower addresses)

...

[amsg(): 100 bytes]
[RA]
[two amsg() arguments]
[interim() stuff, incl. 8000 bytes]
[something else]                        <- current SP
[main() local variables]

...

(upper addresses)
```

[56]Supprimer la vérification de sécurité du buffer

Donc lorsque `main()` appelle `printf()`, elle utilise l'espace de pile où le buffer d'interim() était alloué, et ne zappe pas les 100 octets contenant le message d'erreur, car 8000 octets (ou peut-être bien moins) sont suffisants pour tout ce que `printf()` et les autres fonctions font!

Ça pourrait aussi fonctionner si il y a plusieurs fonctions entre, comme: `main()` → f1() → f2() → f3() ... → amsg(), et alors le résultat de amsg() est utilisé dans `main()`. La distance entre SP dans `main()` et l'adresse de `buf[]` doit être assez grande.

C'est pourquoi les bugs de ce genre sont dangereux: parfois votre code fonctionne (et le bug ne se produit pas), parfois non. Ces genres de bug sont par humour appelés *heisenbugs* ou *schrödinbugs*.

3.29 OpenMP

OpenMP est l'un des moyens les plus simple de paralléliser des algorithmes simples.

À titre d'exemple, essayons de construire un programme pour calculer une *nonce* cryptographique.

Dans mon exemple simpliste, le *nonce* est un nombre ajouté au texte non chiffré afin de produire un hash avec quelques caractéristiques spécifiques.

Par exemple, à certaines étapes, le protocole Bitcoin nécessite de trouver de tels *nonce* dont le hash résultant contient un nombre spécifique de zéros consécutifs. Ceci est aussi appelé «preuve de travail» [57] (i.e., le système prouve qu'il a fait des calculs intensifs et y a passé du temps).

Mon exemple n'est en aucun cas lié au Bitcoin, il va essayer d'ajouter des nombres à la chaîne afin de trouver un nombre tel que le hash de «hello, world!_<number>» avec l'algorithme SHA512, contiendra au moins 3 octets à zéro.

Limitons notre recherche brute-force dans l'intervalle 0..INT32_MAX-1 (i.e., `0x7FFFFFFE` ou 2147483646).

L'algorithme est assez direct:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "sha512.h"

int found=0;
int32_t checked=0;

int32_t* __min;
int32_t* __max;

time_t start;

#ifdef __GNUC__
```

[57] Wikipédia

```
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define max(X,Y) ((X) > (Y) ? (X) : (Y))
#endif

void check_nonce (int32_t nonce)
{
        uint8_t buf[32];
        struct sha512_ctx ctx;
        uint8_t res[64];

        // update statistics
        int t=omp_get_thread_num();

        if (__min[t]==-1)
                __min[t]=nonce;
        if (__max[t]==-1)
                __max[t]=nonce;

        __min[t]=min(__min[t], nonce);
        __max[t]=max(__max[t], nonce);

        // idle if valid nonce found
        if (found)
                return;

        memset (buf, 0, sizeof(buf));
        sprintf (buf, "hello, world!_%d", nonce);

        sha512_init_ctx (&ctx);
        sha512_process_bytes (buf, strlen(buf), &ctx);
        sha512_finish_ctx (&ctx, &res);
        if (res[0]==0 && res[1]==0 && res[2]==0)
        {
                printf ("found (thread %d): [%s]. seconds spent=%d\n", t, ↙
    ↘ buf, time(NULL)-start);
                found=1;
        };
        #pragma omp atomic
        checked++;

        #pragma omp critical
        if ((checked % 100000)==0)
                printf ("checked=%d\n", checked);
};

int main()
{
        int32_t i;
        int threads=omp_get_max_threads();
        printf ("threads=%d\n", threads);

        __min=(int32_t*)malloc(threads*sizeof(int32_t));
        __max=(int32_t*)malloc(threads*sizeof(int32_t));
```

```
        for (i=0; i<threads; i++)
                __min[i]=__max[i]=-1;

        start=time(NULL);

        #pragma omp parallel for
        for (i=0; i<INT32_MAX; i++)
                check_nonce (i);

        for (i=0; i<threads; i++)
                printf ("__min[%d]=0x%08x __max[%d]=0x%08x\n", i, __min[i],↙
    ↘  i, __max[i]);

        free(__min); free(__max);
};
```

La fonction check_nonce() ajoute simplement un nombre à la chaîne, hashe le résultat avec l'algorithme SHA12 et teste si il y a 3 octets à zéro dans le résultat.

Une partie très importante du code est:

```
        #pragma omp parallel for
        for (i=0; i<INT32_MAX; i++)
                check_nonce (i);
```

Oui, c'est simple, sans le #pragma nous appelons check_nonce() pour chaque nombre de 0 à INT32_MAX (0x7fffffff ou 2147483647). Avec le #pragma, le compilateur ajoute du code particulier qui découpe l'intervalle de la boucle en des plus petits, afin de les lancer sur tous les cœurs de CPU disponible[58].

L'exemple peut être compilé[59] dans MSVC 2012:

```
cl openmp_example.c sha512.obj /openmp /O1 /Zi /Faopenmp_example.asm
```

Ou dans GCC:

```
gcc -fopenmp 2.c sha512.c -S -masm=intel
```

3.29.1 MSVC

Maintenant voici comment MSVC 2012 génère la boucle principale:

Listing 3.125 : MSVC 2012

```
        push    OFFSET _main$omp$1
        push    0
        push    1
        call    __vcomp_fork
        add     esp, 16
```

[58]N.B.: Ceci est intentionnellement l'exemple le plus simple possible, mais en pratique, l'utilisation de OpenMP peut être plus difficile et plus complexe.

[59]Les fichiers sha512.(c|h) et u64.h peuvent être pris de la bibliothèque OpenSSL: http://www.openssl.org/source/

Toutes les fonctions préfixées par vcomp sont relatives à OpenMP et sont stockées dans le fichier vcomp*.dll. Donc, ici un groupe de threads est démarré.

Regardons _mainomp1 :

Listing 3.126 : MSVC 2012

```
$T1 = -8        ; size = 4
$T2 = -4        ; size = 4
_main$omp$1 PROC
        push    ebp
        mov     ebp, esp
        push    ecx
        push    ecx
        push    esi
        lea     eax, DWORD PTR $T2[ebp]
        push    eax
        lea     eax, DWORD PTR $T1[ebp]
        push    eax
        push    1
        push    1
        push    2147483646      ; 7ffffffeH
        push    0
        call    __vcomp_for_static_simple_init
        mov     esi, DWORD PTR $T1[ebp]
        add     esp, 24
        jmp     SHORT $LN6@main$omp$1
$LL2@main$omp$1:
        push    esi
        call    _check_nonce
        pop     ecx
        inc     esi
$LN6@main$omp$1:
        cmp     esi, DWORD PTR $T2[ebp]
        jle     SHORT $LL2@main$omp$1
        call    __vcomp_for_static_end
        pop     esi
        leave
        ret     0
_main$omp$1 ENDP
```

Cette fonction va être démarrée n fois en parallèle, où n est le nombre de cœurs du CPU.
vcomp_for_static_simple_init() calcul l'intervalle pour la construction for() du thread courant, dépendant du numéro du thread courant.

Les valeurs de début et de fin sont stockées dans les variables locales $T1 et $T2.
Vous pouvez également remarquer l'argument 7ffffffeh (ou 2147483646) comme argument de la fonction
vcomp_for_static_simple_init()—ceci est le nombre d'itérations de la boucle complète,
qui doit éventuellement être divisée.

Puis nous voyons une nouvelle boucle avec un appel à la fonction check_nonce(),

qui fait tout le travail.

Ajoutons du code au début de la fonction check_nonce() pour collecter des statistiques sur les arguments avec lesquels la fonction a été appelée.

Voici ce que nous pouvons voir lorsque nous le lançons:

```
threads=4
...
checked=2800000
checked=3000000
checked=3200000
checked=3300000
found (thread 3): [hello, world!_1611446522]. seconds spent=3
__min[0]=0x00000000 __max[0]=0x1fffffff
__min[1]=0x20000000 __max[1]=0x3fffffff
__min[2]=0x40000000 __max[2]=0x5fffffff
__min[3]=0x60000000 __max[3]=0x7ffffffe
```

Oui, le résultat est correct, les 3 premiers octets sont des zéros:

```
C:\...\sha512sum test
000000f4a8fac5a4ed38794da4c1e39f54279ad5d9bb3c5465cdf57adaf60403
df6e3fe6019f5764fc9975e505a7395fed780fee50eb38dd4c0279cb114672e2 *test
```

Le temps de traitement est $\approx 2..3$ secondes sur un Intel Xeon E3-1220 3.10 GHz 4-core. Dans le gestionnaire de tâches nous voyons 5 threads: 1 thread principal + 4 autres. Il n'y a pas d'optimisations faites afin de garder cet exemple aussi petit et clair que possible. Mais probablement qu'on pourrait le rendre plus rapide. Mon CPU a 4 cœurs, c'est pourquoi OpenMP a démarré exactement 4 threads.

En regardant la table des statistiques, nous voyons clairement comment la boucle a été découpée en 4 parties égales. Oui bon, presque égales, si nous ne tenons pas compte du dernier bit.

Il y a aussi des pragmas pour les operations atomiques..

Voyons comment ce code est compilé:

```
        #pragma omp atomic
        checked++;

        #pragma omp critical
        if ((checked % 100000)==0)
                printf ("checked=%d\n", checked);
```

Listing 3.127 : MSVC 2012

```
        push    edi
        push    OFFSET _checked
        call    __vcomp_atomic_add_i4
; Line 55
        push    OFFSET _$vcomp$critsect$
        call    __vcomp_enter_critsect
        add     esp, 12
```

```
; Line 56
        mov     ecx, DWORD PTR _checked
        mov     eax, ecx
        cdq
        mov     esi, 100000     ; 000186a0H
        idiv    esi
        test    edx, edx
        jne     SHORT $LN1@check_nonc
; Line 57
        push    ecx
        push    OFFSET ??_C@_0M@NPNHLIOO@checked?$DN?$CFd?6?$AA@
        call    _printf
        pop     ecx
        pop     ecx
$LN1@check_nonc:
        push    DWORD PTR _$vcomp$critsect$
        call    __vcomp_leave_critsect
        pop     ecx
```

Il semble que la fonction `vcomp_atomic_add_i4()` dans vcomp*.dll soit juste une minuscule fonction avec l'instruction `LOCK XADD`[60] dedans.

`vcomp_enter_critsect()` appelle finalement la fonction de l'API win32 `EnterCriticalSection()` [61].

3.29.2 GCC

GCC 4.8.1 produit un programme qui montre exactement la même table de statistique,

donc, l'implémentation de GCC divise la boucle en parties de la même manière.

Listing 3.128 : GCC 4.8.1

```
        mov     edi, OFFSET FLAT:main._omp_fn.0
        call    GOMP_parallel_start
        mov     edi, 0
        call    main._omp_fn.0
        call    GOMP_parallel_end
```

Contrairement à l'implémentation de MSVC, ce que le code de GCC fait est de démarrer 3 threads et lance la quatrième dans le thread courant. Il y a donc 4 threads au lieu de 5 dans MSVC.

Voici les fonctions `main._omp_fn.0` :

Listing 3.129 : GCC 4.8.1

```
main._omp_fn.0:
        push    rbp
        mov     rbp, rsp
```

[60]En savoir plus sur le préfixe LOCK: .1.6 on page 1330
[61]Vous pouvez en lire plus sur les sections critiques ici: 6.5.4 on page 1026

```
        push    rbx
        sub     rsp, 40
        mov     QWORD PTR [rbp-40], rdi
        call    omp_get_num_threads
        mov     ebx, eax
        call    omp_get_thread_num
        mov     esi, eax
        mov     eax, 2147483647 ; 0x7FFFFFFF
        cdq
        idiv    ebx
        mov     ecx, eax
        mov     eax, 2147483647 ; 0x7FFFFFFF
        cdq
        idiv    ebx
        mov     eax, edx
        cmp     esi, eax
        jl      .L15
.L18:
        imul    esi, ecx
        mov     edx, esi
        add     eax, edx
        lea     ebx, [rax+rcx]
        cmp     eax, ebx
        jge     .L14
        mov     DWORD PTR [rbp-20], eax
.L17:
        mov     eax, DWORD PTR [rbp-20]
        mov     edi, eax
        call    check_nonce
        add     DWORD PTR [rbp-20], 1
        cmp     DWORD PTR [rbp-20], ebx
        jl      .L17
        jmp     .L14
.L15:
        mov     eax, 0
        add     ecx, 1
        jmp     .L18
.L14:
        add     rsp, 40
        pop     rbx
        pop     rbp
        ret
```

Ici nous voyons la division clairement: en appelant `omp_get_num_threads()` et `omp_get_thread_num()`

nous obtenons le nombre de threads lancés, le nombre courant de threads, et ainsi détermine l'intervalle de la boucle. Ensuite nous lançons `check_nonce()`.

GCC a également inséré l'instruction `LOCK ADD` directement dans le code, contrairement à MSVC, qui génère un appel à une fonction DLL séparée:

Listing 3.130 : GCC 4.8.1

```
        lock add        DWORD PTR checked[rip], 1
        call    GOMP_critical_start
        mov     ecx, DWORD PTR checked[rip]
        mov     edx, 351843721
        mov     eax, ecx
        imul    edx
        sar     edx, 13
        mov     eax, ecx
        sar     eax, 31
        sub     edx, eax
        mov     eax, edx
        imul    eax, eax, 100000
        sub     ecx, eax
        mov     eax, ecx
        test    eax, eax
        jne     .L7
        mov     eax, DWORD PTR checked[rip]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC2 ; "checked=%d\n"
        mov     eax, 0
        call    printf
.L7:
        call    GOMP_critical_end
```

Les fonctions préfixées par GOMP sont de la bibliothèque GNU OpenMP. Contrairement à vcomp*.dll, son code source est librement disponible: GitHub.

3.30 Division signée en utilisant des décalages

La division non signée par des nombres 2^n est facile, il sufft d'utiliser le décalage de n bit à droite. La division signée par 2^n est aussi facile, mais des corrections doivent être faites avant ou après l'opération de décalage.

D'abord, la plupart des architectures CPU supportent deux opérations de décalage à droite: logique et arithmétique. Lors d'un décalage logique à droite, le bit libre est mis à zéro. C'est SHR en x86. Lors d'un décalage arithmétique à droite, le bit à gauche est mis avec celui qui était à cette position avant le décalage. Ainsi, le signe est conservé lors du décalage. C'est SAR en x86.

Il est intéressant de noter qu'il n'y a pas d'instruction spéciale pour le décalage arithmétique à gauche, car il fonctionne tout simplement comme le décalage logique. Donc, les instructions SAL et SHL en x86 sont mappées sur le même opcode. De nombreux désassembleurs ne connaissent même pas l'instruction SAL et la décode comme SHL.

De ce fait, le décalages arithmétique à droite est utilisé pour les nombres signés. Par exemple, si vous décalez -4 (11111100b) d'1 bit à droite, l'opération de décalage logique produira 01111110b, qui est 126. Le décalage arithmétique à droite produira 11111110b, qui est -2. Jusqu'ici, tout va bien.

Et si nous devons diviser -5 par 2? Ça vaut 2,5 ou juste -2 en arithmétique entière. -5 est 11111011b, en décalant cette valeur de 1 bit à droite, nous obtenons 11111101b,

qui est -3. Ceci est légèrement incorrect.

Un autre exemple: $-\frac{1}{2} = -0.5$ ou 0 en arithmétique entière. Mais -1 est 11111111b, et 11111111b » 1 = 11111111b, qui est encore -1. Ceci est aussi incorrect.

Une solution est d'ajouter 1 à la valeur en entrée si elle est négative.

C'est pourquoi, si nous compilons l'expression x/2, où x est un *signed int*, GCC 4.8 produira quelque chose comme cela:

```
        mov     eax, edi
        shr     eax, 31  ; isoler le bit le plus à gauche, qui est 1 si la valeur est négative et 0 si elle est positive
        add     eax, edi ; ajouter 1 à la valeur en entrée si elle est négative, ne rien faire autrement
        sar     eax      ; décalage arithmétique à droite de un bit
        ret
```

Si vous divisez par 4, il faut ajouter 3 à la valeur en entrée si elle est négative. Donc ceci est ce que GCC 4.8 génère pour x/4 :

```
        lea     eax, [rdi+3] ; préparer la valeur x+3 en avance
        test    edi, edi

        ; si le signe n'est pas négatif (i.e., positif), déplacer la valeur en entrée dans EAX
        ; si le signe est négatif, la valeur x+3 est laissée telle quelle dans EAX
        cmovns  eax, edi
        ; effectuer un décalage arithmétique à droite de 2 bits
        sar     eax, 2
        ret
```

Si vous divisez par 8, il faut ajouter 7 à la valeur en entrée, etc.

MSVC 2013 est légèrement différent. Ceci est la division par 2:

```
        mov     eax, DWORD PTR _a$[esp-4]
        ; étendre la valeur d'entrée à 64-bit dans EDX:EAX
        ; concrètement, cela signifie que EDX sera mis à 0FFFFFFFFh si la valeur en entrée est négative
        ; ... ou à 0 si elle est positive
        cdq
        ; soustraire -1 de la valeur en entrée si elle est négative
        ; ceci est la même chose qu'ajouter 1
        sub     eax, edx
        ; effectuer le décalage arithmétique à droite
        sar     eax, 1
        ret     0
```

La division par 4 dans MSVC 2013 est encore plus complexe:

```
        mov     eax, DWORD PTR _a$[esp-4]
        cdq
        ; maintenant EDX contient 0FFFFFFFFh si la valeur en entrée est négative
```

```
        ; EDX contient 0 si elle est positive
        and     edx, 3
        ; maintenant EDX contient 3 si la valeur en entrée est négative et 0
    autrement
        ; ajouter 3 à la valeur en entrée si elle est négative ou ne rien
    faire autrement:
        add     eax, edx
        ; effectuer le décalage arithmétique à droite
        sar     eax, 2
        ret     0
```

La division par 8 dans MSVC 2013 est similaire, mais 3 bits de EDX sont pris au lieu de 2, produisant une correction de 7 au lieu de 3.

Parfois, Hex-Rays 6.8 ne gère pas correctement un tel code, et il peut produire quelque chose comme ceci:

```
int v0;
...
__int64 v14
...

v14 = ...;
v0 = ((signed int)v14 - HIDWORD(v14)) >> 1;
```

…ce qui peut sans risque être récrit en v0=v14/2.

Hex-Rays 6.8 peut aussi gérer les divisions signées par 4 comme cela:

```
result = ((BYTE4(v25) & 3) + (signed int)v25) >> 2;
```

…peut être récrit en v25 / 4.

En outre, une telle correction est souvent utilisé lorsque la division est remplacée par la multiplication par des *nombres magiques* : lire Mathematical Recipes[62] à propos de la multiplication inverse. Et parfois, un décalage additionnel est utilisé après la multiplication. Par exemple, lorsque GCC optimise $\frac{x}{10}$, il ne peut pas trouver la multiplication inverse pour 10, car l'équation diophantienne n'a pas de solution. Donc il génère du code pour $\frac{x}{5}$ et puis ajoute une opération de décalage arithmétique à droite de 1 bit, pour diviser le résultat par 2. Bien sûr, ceci est seulement vrai pour les entiers signés.

Donc, voici la division par 10 de GCC 4.8:

```
        mov     eax, edi
        mov     edx, 1717986919 ; nombre magique
        sar     edi, 31         ; isoler le bit le plus à gauche (qui reflète
    le signe)
        imul    edx         ; multiplication par le nombre magique (calculer
    x/5)
        sar     edx, 2          ; mainenant calculer (x/5)/2
```

[62]https://math.recipes

```
    ; soustraire -1 (ou ajouter 1) si la valeur en entrée est négative
        ; ne rien faire autrement
        sub     edx, edi
        mov     eax, edx
        ret
```

Résumé: $2^n - 1$ doit être ajouté à la valeur en entrée avant un décalage arithmétique, ou il faut ajouter 1 au résultat final après le décalage. Les deux opérations sont équivalentes l'une à l'autre, donc les développeurs du compilateur doivent choisir ce qui est la plus adapté pour eux. Du point de vue du rétro-ingénieur, cette correction est une preuve manifeste que la valeur a un type signé.

3.31 Un autre heisenbug

Parfois, un tableau (ou tampon) peut déborder à cause d'une *erreur de poteaux et d'intervalles* :

```
#include <stdio.h>

int array1[128];
int important_var1;
int important_var2;
int important_var3;
int important_var4;
int important_var5;

int main()
{
        important_var1=1;
        important_var2=2;
        important_var3=3;
        important_var4=4;
        important_var5=5;

        array1[0]=123;
        array1[128]=456; // BUG

        printf ("important_var1=%d\n", important_var1);
        printf ("important_var2=%d\n", important_var2);
        printf ("important_var3=%d\n", important_var3);
        printf ("important_var4=%d\n", important_var4);
        printf ("important_var5=%d\n", important_var5);
};
```

Ceci est ce que ce programme a affiché dans mon cas (GCC 5.4 x86 sans optimisation sur Linux) :

```
important_var1=1
important_var2=456
important_var3=3
important_var4=4
```

```
important_var5=5
```

Lorsque ça se produit, `important_var2` avait été mise par le compilateur juste après `array1[]` :

Listing 3.131 : objdump -x

```
0804a040 g     O .bss   00000200              array1
...
0804a240 g     O .bss   00000004              important_var2
0804a244 g     O .bss   00000004              important_var4
...
0804a248 g     O .bss   00000004              important_var1
0804a24c g     O .bss   00000004              important_var3
0804a250 g     O .bss   00000004              important_var5
```

D'autres compilateurs peuvent arranger les variables dans un autre ordre, et une autre variable sera écrasée. Ceci est aussi un *heisenbug* (3.28.2 on page 822)—bug qui peut se produire ou passer inaperçu suivant la version du compilateur et les options d'optimisation.

Si toutes les variables et tableaux sont allouées sur la pile locale, la protection de la pile peut être déclenchée, ou pas. Toutefois, Valgrind peut trouver ce genre de bugs.

Un exemple connexe dans le livre (jeu Angband) : 1.27 on page 389.

3.32 Le cas du return oublié

Revoyons la partie "tentative d'utiliser le résultat d'une fonction renvoyant *void*": 1.15.1 on page 145.

Ceci est un bug que j'ai rencontré une fois.

Et c'est encore une autre démonstration de la façon dont C/C++ met les valeurs de retour dans le registre EAX/RAX.

Dans ce bout de code, j'ai oublié d'ajouter `return` :

```
#include <stdio.h>
#include <stdlib.h>

struct color
{
        int R;
        int G;
        int B;
};

struct color* create_color (int R, int G, int B)
{
        struct color* rt=(struct color*)malloc(sizeof(struct color));

        rt->R=R;
```

```
        rt->G=G;
        rt->B=B;
        // must be "return rt;" here
};

int main()
{
        struct color* a=create_color(1,2,3);
        printf ("%d %d %d\n", a->R, a->G, a->B);
};
```

GCC 5.4 sans optimisation le compile silencieusement, sans avertissement. Et le code fonctionne! Voyons pourquoi:

Listing 3.132 : GCC 5.4 sans optimisation

```
create_color:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-24], esi
        mov     DWORD PTR [rbp-28], edx
        mov     edi, 12
        call    malloc
; RAX = pointer to newly allocated buffer
; now fill it with R/G/B:
        mov     QWORD PTR [rbp-8], rax
        mov     rax, QWORD PTR [rbp-8]
        mov     edx, DWORD PTR [rbp-20]
        mov     DWORD PTR [rax], edx
        mov     rax, QWORD PTR [rbp-8]
        mov     edx, DWORD PTR [rbp-24]
        mov     DWORD PTR [rax+4], edx
        mov     rax, QWORD PTR [rbp-8]
        mov     edx, DWORD PTR [rbp-28]
        mov     DWORD PTR [rax+8], edx
        nop
        leave
; RAX hasn't been modified till that point!
        ret
```

Si j'ajoute `return rt;`, seule une instruction est ajoutée à la fin, qui est redondante:

Listing 3.133 : GCC 5.4 sans optimisation

```
create_color:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-24], esi
        mov     DWORD PTR [rbp-28], edx
        mov     edi, 12
```

```
        call    malloc
; RAX = pointer to buffer
        mov     QWORD PTR [rbp-8], rax
        mov     rax, QWORD PTR [rbp-8]
        mov     edx, DWORD PTR [rbp-20]
        mov     DWORD PTR [rax], edx
        mov     rax, QWORD PTR [rbp-8]
        mov     edx, DWORD PTR [rbp-24]
        mov     DWORD PTR [rax+4], edx
        mov     rax, QWORD PTR [rbp-8]
        mov     edx, DWORD PTR [rbp-28]
        mov     DWORD PTR [rax+8], edx
; reload pointer to buffer into RAX again, and this is redundant operation...
        mov     rax, QWORD PTR [rbp-8] ; new instruction
        leave
        ret
```

Des bogues de ce type sont très dangereux, parfois ils apparaissent, parfois ils restent invisibles.

Maintenant, j'essaye GCC avec l'optimisation:

Listing 3.134 : GCC 5.4 avec optimisation

```
create_color:
        rep ret

main:
        xor     eax, eax
; as if create_color() was called and returned 0
        sub     rsp, 8
        mov     r8d, DWORD PTR ds:8
        mov     ecx, DWORD PTR [rax+4]
        mov     edx, DWORD PTR [rax]
        mov     esi, OFFSET FLAT:.LC1
        mov     edi, 1
        call    __printf_chk
        xor     eax, eax
        add     rsp, 8
        ret
```

Le compilateur en déduit que rien n'est renvoyé de la fonction, donc il optimise en enlevant complètement la fonction. Et il suppose que 0 est renvoyé par défaut. Le zéro est utilisé comme une adresse sur une structure dans main().. Bien sûr, ce code plante.

GCC en mode C++ est aussi silencieux à propos de cela.

Essayons MSVC 2015 x86 sans optimisation. Il averti à propos de ce problème:

```
c:\tmp\3.c(19) : warning C4716: 'create_color': must return a value
```

Et il génère du code qui plante:

Listing 3.135 : MSVC 2015 x86 sans optimisation

```
_rt$ = -4
_R$ = 8
_G$ = 12
_B$ = 16
_create_color PROC
        push    ebp
        mov     ebp, esp
        push    ecx
        push    12
        call    _malloc
; EAX = pointer to buffer
        add     esp, 4
        mov     DWORD PTR _rt$[ebp], eax
        mov     eax, DWORD PTR _rt$[ebp]
        mov     ecx, DWORD PTR _R$[ebp]
        mov     DWORD PTR [eax], ecx
        mov     edx, DWORD PTR _rt$[ebp]
        mov     eax, DWORD PTR _G$[ebp]
; EAX is set to G argument:
        mov     DWORD PTR [edx+4], eax
        mov     ecx, DWORD PTR _rt$[ebp]
        mov     edx, DWORD PTR _B$[ebp]
        mov     DWORD PTR [ecx+8], edx
        mov     esp, ebp
        pop     ebp
; EAX = G at this point:
        ret     0
_create_color ENDP
```

Maintenant MSVC 2015 x86 avec optimisation, qui génère du code qui plante aussi mais pour une raison différente.

Listing 3.136 : MSVC 2015 x86 avec optimisation

```
_a$ = -4
_main   PROC
; this is inlined optimized version of create_color():
        push    ecx
        push    12
        call    _malloc
        mov     DWORD PTR [eax], 1
        mov     DWORD PTR [eax+4], 2
        mov     DWORD PTR [eax+8], 3
; EAX points to allocated buffer, and it's filled, OK
; now we reload pointer to buffer, hoping it's in "a" variable
; but inlined function didn't store pointer to "a" variable!
        mov     eax, DWORD PTR _a$[esp+8]
; EAX = some random garbage at this point
        push    DWORD PTR [eax+8]
        push    DWORD PTR [eax+4]
        push    DWORD PTR [eax]
        push    OFFSET $SG6074
        call    _printf
```

```
        xor     eax, eax
        add     esp, 24
        ret     0
_main   ENDP

_R$ = 8
_G$ = 12
_B$ = 16
_create_color PROC
        push    12
        call    _malloc
        mov     ecx, DWORD PTR _R$[esp]
        add     esp, 4
        mov     DWORD PTR [eax], ecx
        mov     ecx, DWORD PTR _G$[esp-4]
        mov     DWORD PTR [eax+4], ecx
        mov     ecx, DWORD PTR _B$[esp-4]
        mov     DWORD PTR [eax+8], ecx
; EAX points to allocated buffer, OK
        ret     0
_create_color ENDP
```

Toutefois, MSVC 2015 x64 sans optimisation génère du code qui fonctionne:

Listing 3.137 : MSVC 2015 x64 sans optimisation

```
rt$ = 32
R$ = 64
G$ = 72
B$ = 80
create_color PROC
        mov     DWORD PTR [rsp+24], r8d
        mov     DWORD PTR [rsp+16], edx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 56
        mov     ecx, 12
        call    malloc
; RAX = allocated buffer
        mov     QWORD PTR rt$[rsp], rax
        mov     rax, QWORD PTR rt$[rsp]
        mov     ecx, DWORD PTR R$[rsp]
        mov     DWORD PTR [rax], ecx
        mov     rax, QWORD PTR rt$[rsp]
        mov     ecx, DWORD PTR G$[rsp]
        mov     DWORD PTR [rax+4], ecx
        mov     rax, QWORD PTR rt$[rsp]
        mov     ecx, DWORD PTR B$[rsp]
        mov     DWORD PTR [rax+8], ecx
        add     rsp, 56
; RAX didn't change down to this point
        ret     0
create_color ENDP
```

MSVC 2015 x64 avec optimisation met le fonction en ligne, comme dans le cas du x86, et le code résultant plante.

Ceci est un morceau de code réel de ma bibliothèque *octothorpe*, qui fonctionnait et dont tous les tests réussissaient. C'était ainsi, sans `return` pendant un certain temps..

```
uint32_t LPHM_u32_hash(void *key)
{
        jenkins_one_at_a_time_hash_u32((uint32_t)key);
}
```

La morale de l'histoire: les warnings sont très importants, utilisez `-Wall`, etc, etc... Lorsque la déclaration `return` est absente, le compilateur peut simplement silencieusement ne rien faire à ce point.

Un tel bug passé inaperçu peut gâcher une journée.

Aussi, *le débogage shotgun* est mauvais, car encore une fois, un tel bogue peut passer inaperçu ("tout fonctionne maintenant, qu'il en soit ainsi, laissons tout ça comme c'est").

Voir aussi: la discussion sur Hacker News[63] et l'article de blog archivée[64].

3.33 Exercice: un peu plus loin avec les pointeur et les unions

Ce code a été copié/collé de *dwm*[65], probablement le plus petit window manager sur Linux de tous les temps.

Le problème: les frappes de l'utilisateur au clavier doivent être réparties aux diverses fonctions de *dwm*. Ceci est en général résolu en utilisant un gros *switch()*.

En général, c'est un problème important qui doit être résolu par tous développeur de jeu vidéo.

Apparemment, le créateur de *dwm* a voulu rendre le code soigné et modifiable par les utilisateurs:

```
...

typedef union {
  int i;
  unsigned int ui;
  float f;
  const void *v;
```

[63] https://news.ycombinator.com/item?id=18671609
[64] https://web.archive.org/web/20190317231721/https://yurichev.com/blog/no_return/
[65] https://dwm.suckless.org/

```
} Arg;

...

typedef struct {
  unsigned int mod;
  KeySym keysym;
  void (*func)(const Arg *);
  const Arg arg;
} Key;

...

static Key keys[] = {
  /* modifier          key        function    argument */
  { MODKEY,            XK_p,      spawn,      {.v = dmenucmd } },
  { MODKEY|ShiftMask,  XK_Return, spawn,      {.v = termcmd } },
  { MODKEY,            XK_b,      togglebar,  {0} },
  { MODKEY,            XK_j,      focusstack, {.i = +1 } },
  { MODKEY,            XK_k,      focusstack, {.i = -1 } },
  { MODKEY,            XK_i,      incnmaster, {.i = +1 } },
  { MODKEY,            XK_d,      incnmaster, {.i = -1 } },
  { MODKEY,            XK_h,      setmfact,   {.f = -0.05} },
  { MODKEY,            XK_l,      setmfact,   {.f = +0.05} },
  { MODKEY,            XK_Return, zoom,       {0} },
  { MODKEY,            XK_Tab,    view,       {0} },
  { MODKEY|ShiftMask,  XK_c,      killclient, {0} },
  { MODKEY,            XK_t,      setlayout,  {.v = &layouts[0]} },
  { MODKEY,            XK_f,      setlayout,  {.v = &layouts[1]} },
  { MODKEY,            XK_m,      setlayout,  {.v = &layouts[2]} },

...

void
spawn(const Arg *arg)
{
...

void
focusstack(const Arg *arg)
{

...
```

Pour chaque touche frappée (ou raccourci), une fonction est définie. Encore mieux: un paramètre (ou argument) peut être passé à une fonction dans chaque cas. Mais les paramètres peuvent avoir des types variés. Donc une *union* est utilisée ici. Une valeur du type requis est mise dans la table. Chaque fonction prend ce dont elle a besoin.

À titre d'exercice, essayez d'écrire un code comme cela, ou plongez-vous dans *dwm* et voyez comment l'*union* est passée aux fonctions et est gérée.

3.34 Windows 16-bit

Les programmes Windows 16-bit sont rares de nos jours, mais ils peuvent être utilisés dans le cadre de rétrocomputing ou d'hacking de dongle (8.8 on page 1081).

Il y a eu des versions 16-bit de Windows jusqu'à la 3.11. 95/98/ME supportaient le code 16-bit, ainsi que les versions 32-bit de la série Windows NT. Les versions 64-bit de Windows NT ne supportaient pas du tout le code exécutable 16-bit.

Le code ressemble a du code MS-DOS.

Les fichiers exécutables sont du type NE (appelé «new executable»).

Tous les exemples considérés ici ont été compilés avec le compilateur OpenWatcom 1.9, en utilisant ces paramètres:

```
wcl.exe -i=C:/WATCOM/h/win/ -s -os -bt=windows -bcl=windows example.c
```

3.34.1 Exemple#1

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        MessageBeep(MB_ICONEXCLAMATION);
        return 0;
};
```

```
WinMain         proc near
                push    bp
                mov     bp, sp
                mov     ax, 30h ; '0' ; MB_ICONEXCLAMATION constant
                push    ax
                call    MESSAGEBEEP
                xor     ax, ax          ; return 0
                pop     bp
                retn    0Ah
WinMain         endp
```

Ça semble facile, jusqu'ici.

3.34.2 Exemple #2

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
```

```
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);
        return 0;
};
```

```
WinMain         proc near
                push    bp
                mov     bp, sp
                xor     ax, ax          ; NULL
                push    ax
                push    ds
                mov     ax, offset aHelloWorld ; 0x18. "hello, world"
                push    ax
                push    ds
                mov     ax, offset aCaption ; 0x10. "caption"
                push    ax
                mov     ax, 3           ; MB_YESNOCANCEL
                push    ax
                call    MESSAGEBOX
                xor     ax, ax          ; return 0
                pop     bp
                retn    0Ah
WinMain         endp

dseg02:0010 aCaption        db 'caption',0
dseg02:0018 aHelloWorld     db 'hello, world',0
```

Quelques points importants ici: la convention d'appel PASCAL impose de passer le premier argument en premier (MB_YESNOCANCEL), et le dernier argument — en dernier (NULL). Cette convention demande aussi à l'appelant de restaurer le pointeur de pile : D'où l'instruction RETN qui a 0Ah comme argument, ce qui implique que le pointeur sera incrémenté de 10 octets lorsque l'on sortira de la fonction. C'est comme stdcall (6.1.2 on page 953), mais les arguments sont passés dans l'ordre «naturel ».

Les pointeurs sont passés par paire: d'abord le segment de données, puis le pointeur dans le segment. Il y a seulemnt un segment dans cet exemple, donc DS pointe toujours sur le segment de données de l'exécutable.

3.34.3 Exemple #3

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    int result=MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL)↙
    ↳ ;
```

```
    if (result==IDCANCEL)
        MessageBox (NULL, "you pressed cancel", "caption", MB_OK);
    else if (result==IDYES)
        MessageBox (NULL, "you pressed yes", "caption", MB_OK);
    else if (result==IDNO)
        MessageBox (NULL, "you pressed no", "caption", MB_OK);

    return 0;
};
```

```
WinMain         proc near
                push    bp
                mov     bp, sp
                xor     ax, ax       ; NULL
                push    ax
                push    ds
                mov     ax, offset aHelloWorld ; "hello, world"
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                mov     ax, 3           ; MB_YESNOCANCEL
                push    ax
                call    MESSAGEBOX
                cmp     ax, 2           ; IDCANCEL
                jnz     short loc_2F
                xor     ax, ax
                push    ax
                push    ds
                mov     ax, offset aYouPressedCanc ; "you pressed cancel"
                jmp     short loc_49
loc_2F:
                cmp     ax, 6           ; IDYES
                jnz     short loc_3D
                xor     ax, ax
                push    ax
                push    ds
                mov     ax, offset aYouPressedYes ; "you pressed yes"
                jmp     short loc_49
loc_3D:
                cmp     ax, 7           ; IDNO
                jnz     short loc_57
                xor     ax, ax
                push    ax
                push    ds
                mov     ax, offset aYouPressedNo ; "you pressed no"
loc_49:
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                xor     ax, ax
```

```
                push    ax
                call    MESSAGEBOX
loc_57:
                xor     ax, ax
                pop     bp
                retn    0Ah
WinMain         endp
```

Exemple un peu plus long de la section précédente.

3.34.4 Exemple #4

```
#include <windows.h>

int PASCAL func1 (int a, int b, int c)
{
        return a*b+c;
};

long PASCAL func2 (long a, long b, long c)
{
        return a*b+c;
};

long PASCAL func3 (long a, long b, long c, int d)
{
        return a*b+c-d;
};

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        func1 (123, 456, 789);
        func2 (600000, 700000, 800000);
        func3 (600000, 700000, 800000, 123);
        return 0;
};
```

```
func1           proc near

c               = word ptr  4
b               = word ptr  6
a               = word ptr  8

                push    bp
                mov     bp, sp
                mov     ax, [bp+a]
                imul    [bp+b]
                add     ax, [bp+c]
                pop     bp
```

```
                retn    6
func1           endp

func2           proc near

arg_0           = word ptr  4
arg_2           = word ptr  6
arg_4           = word ptr  8
arg_6           = word ptr  0Ah
arg_8           = word ptr  0Ch
arg_A           = word ptr  0Eh

                push    bp
                mov     bp, sp
                mov     ax, [bp+arg_8]
                mov     dx, [bp+arg_A]
                mov     bx, [bp+arg_4]
                mov     cx, [bp+arg_6]
                call    sub_B2 ; long 32-bit multiplication
                add     ax, [bp+arg_0]
                adc     dx, [bp+arg_2]
                pop     bp
                retn    12
func2           endp

func3           proc near

arg_0           = word ptr  4
arg_2           = word ptr  6
arg_4           = word ptr  8
arg_6           = word ptr  0Ah
arg_8           = word ptr  0Ch
arg_A           = word ptr  0Eh
arg_C           = word ptr  10h

                push    bp
                mov     bp, sp
                mov     ax, [bp+arg_A]
                mov     dx, [bp+arg_C]
                mov     bx, [bp+arg_6]
                mov     cx, [bp+arg_8]
                call    sub_B2 ; long 32-bit multiplication
                mov     cx, [bp+arg_2]
                add     cx, ax
                mov     bx, [bp+arg_4]
                adc     bx, dx          ; BX=high part, CX=low part
                mov     ax, [bp+arg_0]
                cwd                     ; AX=low part d, DX=high part d
                sub     cx, ax
                mov     ax, cx
                sbb     bx, dx
                mov     dx, bx
                pop     bp
```

```
                retn    14
func3           endp

WinMain         proc near
                push    bp
                mov     bp, sp
                mov     ax, 123
                push    ax
                mov     ax, 456
                push    ax
                mov     ax, 789
                push    ax
                call    func1
                mov     ax, 9      ; high part of 600000
                push    ax
                mov     ax, 27C0h  ; low part of 600000
                push    ax
                mov     ax, 0Ah    ; high part of 700000
                push    ax
                mov     ax, 0AE60h ; low part of 700000
                push    ax
                mov     ax, 0Ch    ; high part of 800000
                push    ax
                mov     ax, 3500h  ; low part of 800000
                push    ax
                call    func2
                mov     ax, 9      ; high part of 600000
                push    ax
                mov     ax, 27C0h  ; low part of 600000
                push    ax
                mov     ax, 0Ah    ; high part of 700000
                push    ax
                mov     ax, 0AE60h ; low part of 700000
                push    ax
                mov     ax, 0Ch    ; high part of 800000
                push    ax
                mov     ax, 3500h  ; low part of 800000
                push    ax
                mov     ax, 7Bh    ; 123
                push    ax
                call    func3
                xor     ax, ax     ; return 0
                pop     bp
                retn    0Ah
WinMain         endp
```

Les valeurs 32-bit (le type de donnée `long` implique 32 bits, tandis que *int* est 16-bit en code 16-bit (à la fois pour MS-DOS et Win16) sont passées par paires. C'est tout comme lorsqu'une valeur 64-bit est utilisée dans un environnement 32-bit (1.34 on page 509).

sub_B2 voici une fonction de bibliothèques écrite par les développeurs du compilateurs qui fait la «multiplication des long» (i.e., multiplie deux valeurs 32-bits).

D’autres fonctions de compilateur qui font la même chose sont listées ici: .5 on page 1354, .4 on page 1354.

La paire d’instructions ADD/ADC est utilisée pour l’addition de valeurs composées: ADD peut mettre le flag CF à 0/1, et ADC l’utilise après.

La paire d’instructions SUB/SBB est utilisée pour la soustraction: SUB peut mettre la flag CF à 0/1, et SBB l’utilise après.

Les valeurs 32-bit sont renvoyées de la fonction dans la paire de registres DX:AX.

Les constantes sont aussi passées par paires dans WinMain() ici.

La constante 123 typée *int* est d’abord converti suivant le signe de la valeur 32-bit en utilisant l’instruction CWD.

3.34.5 Exemple #5

```
#include <windows.h>

int PASCAL string_compare (char *s1, char *s2)
{
        while (1)
        {
                if (*s1!=*s2)
                        return 0;
                if (*s1==0 || *s2==0)
                        return 1; // end of string
                s1++;
                s2++;
        };

};

int PASCAL string_compare_far (char far *s1, char far *s2)
{
        while (1)
        {
                if (*s1!=*s2)
                        return 0;
                if (*s1==0 || *s2==0)
                        return 1; // end of string
                s1++;
                s2++;
        };

};

void PASCAL remove_digits (char *s)
{
        while (*s)
        {
                if (*s>='0' && *s<='9')
                        *s='-';
```

```
                s++;
        };
};

char str[]="hello 1234 world";

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        string_compare ("asd", "def");
        string_compare_far ("asd", "def");
        remove_digits (str);
        MessageBox (NULL, str, "caption", MB_YESNOCANCEL);
        return 0;
};
```

```
string_compare  proc near

arg_0 = word ptr  4
arg_2 = word ptr  6

      push    bp
      mov     bp, sp
      push    si
      mov     si, [bp+arg_0]
      mov     bx, [bp+arg_2]

loc_12: ; CODE XREF: string_compare+21j
      mov     al, [bx]
      cmp     al, [si]
      jz      short loc_1C
      xor     ax, ax
      jmp     short loc_2B


loc_1C: ; CODE XREF: string_compare+Ej
      test    al, al
      jz      short loc_22
      jnz     short loc_27

loc_22: ; CODE XREF: string_compare+16j
      mov     ax, 1
      jmp     short loc_2B


loc_27: ; CODE XREF: string_compare+18j
      inc     bx
      inc     si
      jmp     short loc_12
```

```
loc_2B: ; CODE XREF: string_compare+12j
        ; string_compare+1Dj
      pop     si
      pop     bp
      retn    4
string_compare  endp

string_compare_far proc near ; CODE XREF: WinMain+18p

arg_0 = word ptr  4
arg_2 = word ptr  6
arg_4 = word ptr  8
arg_6 = word ptr  0Ah

      push    bp
      mov     bp, sp
      push    si
      mov     si, [bp+arg_0]
      mov     bx, [bp+arg_4]

loc_3A: ; CODE XREF: string_compare_far+35j
      mov     es, [bp+arg_6]
      mov     al, es:[bx]
      mov     es, [bp+arg_2]
      cmp     al, es:[si]
      jz      short loc_4C
      xor     ax, ax
      jmp     short loc_67


loc_4C: ; CODE XREF: string_compare_far+16j
      mov     es, [bp+arg_6]
      cmp     byte ptr es:[bx], 0
      jz      short loc_5E
      mov     es, [bp+arg_2]
      cmp     byte ptr es:[si], 0
      jnz     short loc_63

loc_5E: ; CODE XREF: string_compare_far+23j
      mov     ax, 1
      jmp     short loc_67


loc_63: ; CODE XREF: string_compare_far+2Cj
      inc     bx
      inc     si
      jmp     short loc_3A


loc_67: ; CODE XREF: string_compare_far+1Aj
        ; string_compare_far+31j
      pop     si
      pop     bp
```

```
      retn    8
string_compare_far endp

remove_digits   proc near ; CODE XREF: WinMain+1Fp

arg_0 = word ptr  4

      push    bp
      mov     bp, sp
      mov     bx, [bp+arg_0]

loc_72: ; CODE XREF: remove_digits+18j
      mov     al, [bx]
      test    al, al
      jz      short loc_86
      cmp     al, 30h ; '0'
      jb      short loc_83
      cmp     al, 39h ; '9'
      ja      short loc_83
      mov     byte ptr [bx], 2Dh ; '-'

loc_83: ; CODE XREF: remove_digits+Ej
        ; remove_digits+12j
      inc     bx
      jmp     short loc_72


loc_86: ; CODE XREF: remove_digits+Aj
      pop     bp
      retn    2
remove_digits   endp

WinMain proc near ; CODE XREF: start+EDp
      push    bp
      mov     bp, sp
      mov     ax, offset aAsd ; "asd"
      push    ax
      mov     ax, offset aDef ; "def"
      push    ax
      call    string_compare
      push    ds
      mov     ax, offset aAsd ; "asd"
      push    ax
      push    ds
      mov     ax, offset aDef ; "def"
      push    ax
      call    string_compare_far
      mov     ax, offset aHello1234World ; "hello 1234 world"
      push    ax
      call    remove_digits
      xor     ax, ax
      push    ax
      push    ds
```

```
	mov	ax, offset aHello1234World ; "hello 1234 world"
	push	ax
	push	ds
	mov	ax, offset aCaption ; "caption"
	push	ax
	mov	ax, 3 ; MB_YESNOCANCEL
	push	ax
	call	MESSAGEBOX
	xor	ax, ax
	pop	bp
	retn	0Ah
WinMain endp
```

Nous voyons ici une différence entre les pointeurs appelés «near» et «far» : un autre effet bizarre de la mémoire segmentée en 16-bit 8086.

Vous pouvez en lire plus à ce sujet ici: **??** on page ??.

Les pointeurs «near» sont ceux qui pointent dans le segment de données courant. C'est pourquoi la fonction `string_compare()` prend seulement deux pointeurs 16-bit, et accède des données dans le segment sur lequel DS pointe (L'instruction `mov al, [bx]` fonctionne en fait comme `mov al, ds:[bx]`—DS est implicite ici).

Les pointeurs «far» sont ceux qui pointent sur des données dans un autre segment de mémoire. C'est pourquoi `string_compare_far()` prend la paire de 16-bit comme un pointeur, charge la partie haute dans le registre de segment ES et accède aux données à travers lui (`mov al, es:[bx]`). Les pointeurs «far» sont aussi utilisés dans mon exemple win16 `MessageBox()` : 3.34.2 on page 840. En effet, le noyau de Windows n'est pas au courant du segment de données qui doit être utilisé pour accéder aux chaînes de texte, donc il a besoin de l'information complète. La raison de cette distinction est qu'un programme compact peut n'utiliser qu'un segment de données de 64kb, donc il n'a pas besoin de passer la partie haute de l'adresse, qui est toujours la même. Un programme plus gros peut utiliser plusieurs segments de données de 64kb, donc il doit spécifier le segment de données à chaque fois.

C'est la même histoire avec les segments de code. Un programme compact peut avoir tout son code exécutable dans un seul segment de 64kb, donc toutes les fonctions y seront appelées en utilisant l'instruction CALL NEAR, et le contrôle du flux sera renvoyé en utilisant RETN. Mais si il y a plusieurs segments de code, alors l'adresse d'une fonction devra être spécifiée par une paire, et sera appelée en utilisant l'instruction CALL FAR, et le contrôle du flux renvoyé en utilisant RETF.

Ceci est ce qui est mis dans le compilateur en spécifiant le «modèle de mémoire».

Les compilateurs qui ciblent MS-DOS et Win16 ont des bibliothèques spécifiques pour chaque modèle de mémoire: elles diffèrent par le type de pointeurs pour le code et les données.

3.34.6 Exemple #6

```
#include <windows.h>
#include <time.h>
#include <stdio.h>
```

```
char strbuf[256];

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{

        struct tm *t;
        time_t unix_time;

        unix_time=time(NULL);

        t=localtime (&unix_time);

        sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900, ↙
    ↘ t->tm_mon, t->tm_mday,
                t->tm_hour, t->tm_min, t->tm_sec);

        MessageBox (NULL, strbuf, "caption", MB_OK);
        return 0;
};
```

```
WinMain         proc near

var_4           = word ptr -4
var_2           = word ptr -2

                push    bp
                mov     bp, sp
                push    ax
                push    ax
                xor     ax, ax
                call    time_
                mov     [bp+var_4], ax    ; low part of UNIX time
                mov     [bp+var_2], dx    ; high part of UNIX time
                lea     ax, [bp+var_4]    ; take a pointer of high part
                call    localtime_
                mov     bx, ax            ; t
                push    word ptr [bx]     ; second
                push    word ptr [bx+2]   ; minute
                push    word ptr [bx+4]   ; hour
                push    word ptr [bx+6]   ; day
                push    word ptr [bx+8]   ; month
                mov     ax, [bx+0Ah]      ; year
                add     ax, 1900
                push    ax
                mov     ax, offset a04d02d02d02d02 ;
    "%04d-%02d-%02d %02d:%02d:%02d"
                push    ax
                mov     ax, offset strbuf
                push    ax
                call    sprintf_
```

```
                add     sp, 10h
                xor     ax, ax           ; NULL
                push    ax
                push    ds
                mov     ax, offset strbuf
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                xor     ax, ax          ; MB_OK
                push    ax
                call    MESSAGEBOX
                xor     ax, ax
                mov     sp, bp
                pop     bp
                retn    0Ah
WinMain         endp
```

Le temps UNIX est une valeur 32-bit, donc il est renvoyé dans la paire de registres `DX:AX` et est stocké dans deux variables locales 16-bit. Puis, un pointeur sur la paire est passé à la fonction `localtime()`. La fonction `localtime()` a une structure `struct tm` allouée quelque part dans les entrailles de la bibliothèque C, donc seul un pointeur est renvoyé.

À propos, ceci implique aussi que la fonction ne peut pas être appelée tant que le résultat n'a pas été utilisé.

Pour les fonctions `time()` et `localtime()`, une convention d'appel Watcom est utilisée ici: les quatre premiers arguments sont passés dans les registres AX, DX, BX et CX, et le reste des arguments par la pile.

Les fonctions utilisant cette convention sont aussi marquées par un souligné à la fin de leur nom.

`sprintf()` n'utilise pas la convention d'appel PASCAL, ni la Watcom,
donc les arguments sont passés de la manière *cdecl* normale (6.1.1 on page 953).

Variables globales

Ceci est le même exemple, mais cette fois les variables sont globales:

```
#include <windows.h>
#include <time.h>
#include <stdio.h>

char strbuf[256];
struct tm *t;
time_t unix_time;

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
```

```
        unix_time=time(NULL);

        t=localtime (&unix_time);

        sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900, ↙
    ↘ t->tm_mon, t->tm_mday,
                t->tm_hour, t->tm_min, t->tm_sec);

        MessageBox (NULL, strbuf, "caption", MB_OK);
        return 0;
};
```

```
unix_time_low   dw 0
unix_time_high  dw 0
t               dw 0

WinMain         proc near
                push    bp
                mov     bp, sp
                xor     ax, ax
                call    time_
                mov     unix_time_low, ax
                mov     unix_time_high, dx
                mov     ax, offset unix_time_low
                call    localtime_
                mov     bx, ax
                mov     t, ax               ; will not be used in future...
                push    word ptr [bx]       ; seconds
                push    word ptr [bx+2]     ; minutes
                push    word ptr [bx+4]     ; hour
                push    word ptr [bx+6]     ; day
                push    word ptr [bx+8]     ; month
                mov     ax, [bx+0Ah]        ; year
                add     ax, 1900
                push    ax
                mov     ax, offset a04d02d02d02d02 ;
    "%04d-%02d-%02d %02d:%02d:%02d"
                push    ax
                mov     ax, offset strbuf
                push    ax
                call    sprintf_
                add     sp, 10h
                xor     ax, ax          ; NULL
                push    ax
                push    ds
                mov     ax, offset strbuf
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                xor     ax, ax          ; MB_OK
                push    ax
                call    MESSAGEBOX
```

```
                xor     ax, ax          ; return 0
                pop     bp
                retn    0Ah
WinMain         endp
```

t ne va pas être utilisée, mais le compilateur a généré le code qui stocke la valeur. Car il n'est pas sûr, peut-être que la valeur sera utilisée dans un autre module.

Chapitre 4

Java

4.1 Java

4.1.1 Introduction

Il existe des décompilateurs très connus pour Java (ou pour du bytecode JVM en général) [1].

La raison est que la décompilation du bytecode JVM est un peu plus facile que du code x86 de plus bas niveau:

- Il y a bien plus d'informations sur les types de données.
- Le modèle de la mémoire JVM est beaucoup plus rigoureux et décrit.
- Le compilateur Java ne fait pas d'optimisation (JVM JIT[2] le fait à l'exécution), donc le bytecode dans les fichiers de classe est généralement assez lisible.

Quand est-ce que la connaissance de la JVM est utile ?

- Créer des patchs "Quick-and-dirty" des fichiers de classe sans avoir besoin de recompiler les résultats du décompilateur.
- Analyse de code obfusqué.
- Analyser du code généré par les nouveaux compilateurs Java, pour lesquels il n'existe pas encore de décompilateur mis à jour.
- Construire votre propre obfuscateur.
- Construire un compilateur générateur de code (back-end) ciblant la JVM (comme Scala, Clojure, etc. [3]).

Commençons avec quelques bouts de code. Le JDK 1.7 est ici utilisé partout, sauf mention contraire.

[1] Par exemple, JAD: http://varaneckas.com/jad/
[2] Just-In-Time compilation
[3] Liste complète: http://en.wikipedia.org/wiki/List_of_JVM_languages

Voici la commande utilisée partout pour décompiler les fichiers de classe : `javap -c -verbose`.

Voici le livre que j'ai utilisé pour préparer tous les exemples : [Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] [4].

4.1.2 Renvoyer une valeur

La fonction Java la plus simple est probablement celle qui renvoie une valeur.

Il faut garder en tête qu'il n'y a pas de fonction «libre» en Java au sens commun, ce sont des «méthodes».

Chaque méthode est liée à une classe, donc il n'est pas possible de définir une méthode en dehors d'une classe.

Mais nous allons quand même les appeler des «fonctions», par simplicité.

```
public class ret
{
        public static int main(String[] args)
        {
                return 0;
        }
}
```

Compilons le :

```
javac ret.java
```

...et décompilons le en utilisant l'outil Java standard :

```
javap -c -verbose ret.class
```

Nous obtenons :

Listing 4.1 : JDK 1.7 (extrait)

```
  public static int main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iconst_0
         1: ireturn
```

Les développeurs Java ont décidé que comme 0 est l'une des constantes les plus utilisées en programmation, alors il existe une courte instruction séparée d'un octet, `iconst_0` qui pousse 0 [5].

Il y a aussi `iconst_1` (qui pousse 1), `iconst_2`, etc., jusqu'à `iconst_5`.

[4] Aussi disponible en https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf ; http://docs.oracle.com/javase/specs/jvms/se7/html/

[5] Comme en MIPS où un registre séparé existe pour la constante zéro : 1.5.4 on page 36.

Il existe également l'instruction `iconst_m1` qui pousse -1.

La pile est utilisée en JVM pour passer des données à une fonction appelée et également pour renvoyer des valeurs. Donc `iconst_0` pousse 0 sur la pile. `ireturn` renvoie une valeur entière (*i* dans le nom signifie *integer*) depuis le TOS[6].

Réécrivons légèrement notre exemple, pour qu'il renvoie 1234 maintenant :

```
public class ret
{
        public static int main(String[] args)
        {
                return 1234;
        }
}
```

...nous avons :

Listing 4.2 : JDK 1.7 (extrait)

```
  public static int main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: sipush        1234
         3: ireturn
```

`sipush` (*short integer*) pousse 1234 sur la pile. *short* dans le nom implique qu'une valeur de 16-bit va être poussée. Le nombre 1234 tiens bien, en effet, dans une valeur de 16-bit.

Qu'en est-il des valeurs plus grandes ?

```
public class ret
{
        public static int main(String[] args)
        {
                return 12345678;
        }
}
```

Listing 4.3 : Constant pool

```
...
   #2 = Integer            12345678
...
```

```
  public static int main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: ldc           #2                  // int 12345678
         2: ireturn
```

[6] Top of Stack

Ce n’est pas possible d’encoder un nombre 32-bit dans une instruction opcode JVM, les développeurs n’ont pas laissé une telle possibilité.

Donc le nombre 32-bit 12345678 est enregistré dans ce qu’on appelle le «constant pool» qui est, disons, la bibliothèque des constantes les plus utilisées (incluant les strings, objects, etc.).

Cette façon de passer des constantes n’est pas propre à JVM.

MIPS, ARM et les autres CPUs RISC ne peuvent pas non plus encoder un nombre 32-bit dans un opcode de 32-bit, donc le code CPU RISC (incluant MIPS et ARM) doit construire la valeur en plusieurs étapes, ou le garder dans le segment des données : 1.39.3 on page 570, 1.40.1 on page 574.

Le code MIPS a aussi traditionnellement un pool des constantes, nommé «literal pool», les segments sont nommés «.lit4» (pour des nombres flottants constants de simples précisions sur 32-bit) et «.lit8» (pour des nombres flottants constants de double précision sur 64-bit).

Essayons quelques autres types de données !

Boolean:

```
public class ret
{
        public static boolean main(String[] args)
        {
                return true;
        }
}
```

```
  public static boolean main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iconst_1
         1: ireturn
```

Ce bytecode JVM n’est pas différent de celui qui retourne l’entier 1.

Les emplacements de données 32-bits dans la pile sont aussi utilisés ici pour des valeurs booléennes, comme en C/C++.

Mais on ne peut pas utiliser la valeur booléenne retournée comme un entier ou vice versa – l’information du type est enregistrée dans le fichier de classe et et vérifiée à l’exécution.

C’est la même histoire qu’un *short* 16-bit :

```
public class ret
{
        public static short main(String[] args)
        {
                return 1234;
        }
}
```

```
  public static short main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: sipush        1234
         3: ireturn
```

…et *char* !

```
public class ret
{
        public static char main(String[] args)
        {
                return 'A';
        }
}
```

```
  public static char main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: bipush        65
         2: ireturn
```

bipush signifie «push byte». Inutile de préciser qu'un *char* en Java est un caractère UTF-16 16-bit, ce qui équivaut à un *short*, mais le code ASCII du caractère «A» est 65, et c'est possible d'utiliser cette instruction pour pousser un octet dans la pile.

Essayons aussi un *byte* :

```
public class retc
{
        public static byte main(String[] args)
        {
                return 123;
        }
}
```

```
  public static byte main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: bipush        123
         2: ireturn
```

On peut se demander, pourquoi s'embêter avec avec un type de donnée *short* de 16-bit qui fonctionne en interne comme un entier 32-bit ?

Pourquoi utiliser un type de donnée *char* si c'est pareil qu'un type de donnée *short* ?

La réponse est simple : pour le contrôle du type de données et pour la lisibilité du code source.

Un *char* peut être essentiellement le même qu'un *short*, mais nous saisissons rapidement que c'est un substitut pour un caractère 16-bit, et non pour une autre valeur entière.

Quand on utilise *short*, nous montrons à tout le monde que la plage de la variable est limitée à 16 bits.

C'est une très bonne idée d'utiliser le type *boolean* où c'est nécessaire, plutôt que le *int* de style C.

Il y a aussi un type de donnée entier sur 64-bits en Java :

```
public class ret3
{
        public static long main(String[] args)
        {
                return 1234567890123456789L;
        }
}
```

Listing 4.4 : Constant pool

```
...
   #2 = Long               1234567890123456789l
...
```

```
  public static long main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: ldc2_w        #2                  // long 1234567890123456789l
         3: lreturn
```

Le nombre 64-bit est aussi stocké dans le pool des constantes, `ldc2_w` le charge et `lreturn` (*long return*) le retourne.

L'instruction `ldc2_w` est aussi utilisée pour charger des nombres flottants double précision (qui occupent aussi 64 bits) depuis le pool des constantes :

```
public class ret
{
        public static double main(String[] args)
        {
                return 123.456d;
        }
}
```

Listing 4.5 : Constant pool

```
...
   #2 = Double             123.456d
...
```

```
  public static double main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: ldc2_w        #2                  // double 123.456d
         3: dreturn
```

`dreturn` signifie «return double».

Et enfin, un nombre flottant simple précision :

```
public class ret
{
        public static float main(String[] args)
        {
                return 123.456f;
        }
}
```

Listing 4.6 : Constant pool

```
...
   #2 = Float              123.456f
...
```

```
  public static float main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: ldc           #2                  // float 123.456f
         2: freturn
```

L'instruction `ldc` utilisée ici est la même que celle pour charger des nombres entiers de 32-bit depuis le pool des constantes.

`freturn` signifie «return float».

Maintenant, qu'en est-il de la fonction qui ne retourne rien ?

```
public class ret
{
        public static void main(String[] args)
        {
                return;
        }
}
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=0, locals=1, args_size=1
         0: return
```

Cela signifie que l'instruction `return` est utilisée pour retourner le contrôle sans renvoyer une vraie valeur.

En sachant cela, il est très facile de déduire le type renvoyé par des fonctions (ou des méthodes) depuis la dernière instruction.

4.1.3 Fonctions de calculs simples

Continuons avec une fonction de calcul simple.

```
public class calc
{
        public static int half(int a)
        {
                return a/2;
        }
}
```

Voici la sortie quand l'instruction `iconst_2` est utilisée :

```
  public static int half(int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: iload_0
         1: iconst_2
         2: idiv
         3: ireturn
```

`iload_0` prend le zéroième argument de la fonction et le pousse sur la pile.

`iconst_2` pousse 2 sur la pile. Après l'exécution de ces deux instructions, voici à quoi ressemble la pile :

```
      +---+
TOS ->| 2 |
      +---+
      | a |
      +---+
```

`idiv` prend juste les deux valeurs depuis le TOS, divise l'un par l'autre et laisse le résultat au TOS :

```
      +--------+
TOS ->| result |
      +--------+
```

`ireturn` le prend et le renvoie.

Procédons avec un nombre flottant double précision :

```
public class calc
{
        public static double half_double(double a)
```

```
        {
                return a/2.0;
        }
}
```

Listing 4.7 : Constant pool

```
...
   #2 = Double             2.0d
...
```

```
  public static double half_double(double);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=2, args_size=1
         0: dload_0
         1: ldc2_w        #2                  // double 2.0d
         4: ddiv
         5: dreturn
```

C'est pareil, mais l'instruction `ldc2_w` est utilisée pour charger la constante 2.0 depuis le pool des constantes.

Aussi, les trois autres instructions sont préfixées par *d*, ce qui signifie qu'elles travaillent avec des valeurs de type *double*.

Utilisons maintenant une fonction avec deux arguments :

```
public class calc
{
        public static int sum(int a, int b)
        {
                return a+b;
        }
}
```

```
  public static int sum(int, int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=2
         0: iload_0
         1: iload_1
         2: iadd
         3: ireturn
```

`iload_0` charge le premier argument de la fonction (a), `iload_1`—le second (b).

Voici la pile après l'exécution de ces instructions :

```
      +---+
TOS ->| b |
      +---+
      | a |
      +---+
```

iadd ajoute les deux valeurs et laisse le résultat au TOS :

```
      +----------+
TOS ->| resultat |
      +----------+
```

Étendons cet exemple avec le type *long* :

```
        public static long lsum(long a, long b)
        {
                return a+b;
        }
```

...nous avons :

```
  public static long lsum(long, long);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=4, args_size=2
         0: lload_0
         1: lload_2
         2: ladd
         3: lreturn
```

La deuxième instruction lload prend le second argument depuis la 2ème position.

C'est parce que la valeur d'un *long* de 64-bit occupe exactement deux places de 32-bit.

Exemple un peu plus avancé :

```
public class calc
{
        public static int mult_add(int a, int b, int c)
        {
                return a*b+c;
        }
}
```

```
  public static int mult_add(int, int, int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=3, args_size=3
         0: iload_0
         1: iload_1
         2: imul
         3: iload_2
         4: iadd
         5: ireturn
```

La première étape est la multiplication. Le produit est laissé au TOS :

```
      +---------+
TOS ->| produit |
      +---------+
```

iload_2 charge le troisième argument (c) dans la pile:

```
      +---------+
TOS ->|    c    |
      +---------+
      | produit |
      +---------+
```

Maintenant l'instruction iadd peut ajouter les deux valeurs.

4.1.4 Modèle de mémoire de la JVM

x86 et d'autres environnements de bas niveau utilisent la pile pour le passage des paramètres et le stockage de variables locales.

La JVM est légèrement différente.

Elle a:

- Le tableau des variables locales, Local Variable Array (LVA[7]). Il est utilisé comme stockage pour les paramètres en entrée de fonction et les variables locales.

  Des instructions comme iload_0 charge une valeur depuis cet espace.

  istore y stocke des valeurs. Au début les paramètres de la fonction sont stockés: commençant à 0 ou à 1 (si l'indice 0 est occupé par le pointeur *this*).

  Ensuite les variables locales sont allouées.

  Chaque slot a une taille de 32-bit.

  De ce fait, les valeurs de types de données *long* et *double* occupent deux slots.

- Pile des opérandes (ou simplement «pile»). Elle est utilisée pour les calculs et la passage de paramètres lors de l'appel d'autres fonctions.

  Contrairement aux environnements bas niveau comme x86, il n'est pas possible d'accèder à la pile sans utiliser des instructions qui poussent ou prennent des valeurs dans/depuis la pile.

- Heap. Il est utilisé pour le stockage d'objets et de tableaux.

Ces 3 espaces sont isolés les uns des autres.

4.1.5 Appel de fonction simple

Math.random() renvoie un nombre pseudo-aléatoire dans l'intervalle [0.0 …1.0], mais disons que une certaine raison, nous devons concevoir une fonction qui renvoie un nombre dans l'intervalle [0.0 …0.5] :

```
public class HalfRandom
{
        public static double f()
        {
```

[7] (Java) Local Variable Array

```
                return Math.random()/2;
        }
}
```

Listing 4.8 : Constant pool

```
...
   #2 = Methodref          #18.#19    //  java/lang/Math.random:()D
   #3 = Double             2.0d
...
  #12 = Utf8               ()D
...
  #18 = Class              #22        //  java/lang/Math
  #19 = NameAndType        #23:#12    //  random:()D
  #22 = Utf8               java/lang/Math
  #23 = Utf8               random
```

```
  public static double f();
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=0, args_size=0
         0: invokestatic  #2          // Method java/lang/Math.random:()D
         3: ldc2_w        #3          // double 2.0d
         6: ddiv
         7: dreturn
```

`invokestatic` appelle la fonction `Math.random()` et laisse le résultat sur le TOS.

Le résultat est divisé par 2.0 et renvoyé.

Mais comment est encodé le nom de la fonction?

Il est encodé dans le pool constant en utilisant une expression `Methodref`.

Il définit les noms de classe et méthode.

Le premier champ de `Methodref` pointe sur une expression `Class` qui, à son tour, pointe sur la chaîne de texte usuel («java/lang/Math»).

La seconde expression de `Methodref` pointe sur une expression NameAndType qui a aussi deux liens sur des chaînes.

La première chaîne est «random», qui est le nom de la méthode.

La seconde chaîne est «()D», qui encode le type de la fonction. Cela signifie qu'elle renvoie une valeur *double* (d'où le *D* dans la chaîne).

Ceci est la façon dont 1) la JVM peut vérifier la justesse des types de données; 2) les décompilateurs Java peuvent retrouver les types de données depuis un fichier de classe compilée.

Maintenant, essayons l'exemple «Hello, world!» :

```
public class HelloWorld
{
        public static void main(String[] args)
```

```
        {
                System.out.println("Hello, World");
        }
}
```

Listing 4.9 : Constant pool

```
...
   #2 = Fieldref           #16.#17        //  java/lang/System.out:Ljava/io↙
    ↳ /PrintStream;
   #3 = String             #18            //  Hello, World
   #4 = Methodref          #19.#20        //  java/io/PrintStream.println:(↙
    ↳ Ljava/lang/String;)V
...
  #16 = Class              #23            //  java/lang/System
  #17 = NameAndType        #24:#25        //  out:Ljava/io/PrintStream;
  #18 = Utf8               Hello, World
  #19 = Class              #26            //  java/io/PrintStream
  #20 = NameAndType        #27:#28        //  println:(Ljava/lang/String;)V
...
  #23 = Utf8               java/lang/System
  #24 = Utf8               out
  #25 = Utf8               Ljava/io/PrintStream;
  #26 = Utf8               java/io/PrintStream
  #27 = Utf8               println
  #28 = Utf8               (Ljava/lang/String;)V
...
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2        // Field java/lang/System.out:Ljava/io/↙
    ↳ PrintStream;
         3: ldc           #3        // String Hello, World
         5: invokevirtual #4        // Method java/io/PrintStream.println:(↙
    ↳ Ljava/lang/String;)V
         8: return
```

`ldc` à l'offset 3 prend un pointeur sur la chaîne «Hello, World» dans le pool constant et le pousse sur la pile.

C'est appelé une *référence* dans le monde Java, mais c'est plutôt un pointeur, ou une adresse[8].

L'instruction connue `invokevirtual` prend les informations concernant la fonction `println` (ou méthode) depuis le pool constant et l'appelle.

Comme on peut le savoir, il y a plusieurs méthodes `println`, une pour chaque type de données.

Dans notre cas, c'est la version de `println` destinée au type de données *String*.

[8]À propos de la différence entre pointeurs et *références* en C++ voir: 3.21.3 on page 721.

Mais qu'en est-il de la première instruction `getstatic` ?

Cette instruction prend une *référence* (ou l'adresse de) un champ de l'objet `System.out` et le pousse sur la pile.

Cette valeur se comporte comme le pointeur *this* pour la méthode `println`.

Ainsi, en interne, la méthode `println` prend deux paramètres en entrée: 1)*this*, i.e., un pointeur sur un objet; 2) l'adresse de la chaîne «Hello, World ».

En effet, `println()` est appelé comme une méthode dans un objet `System.out` initialisé.

Par commodité, l'utilitaire `javap` écrit toutes ces informations dans les commentaires.

4.1.6 Appel de beep()

Ceci est un simple appel de deux fonctions sans paramètre:

```
	public static void main(String[] args)
	{
		java.awt.Toolkit.getDefaultToolkit().beep();
	};
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: invokestatic  #2      // Method java/awt/Toolkit.↙
    ↳ getDefaultToolkit:()Ljava/awt/Toolkit;
         3: invokevirtual #3      // Method java/awt/Toolkit.beep:()V
         6: return
```

Le premier `invokestatic` à l'offset 0 appelle
`java.awt.Toolkit.getDefaultToolkit()`, qui renvoie une référence sur un objet de la classe `Toolkit`.
L'instruction `invokevirtual` à l'offset 3 appelle la méthode `beep()` de cette classe.

4.1.7 Congruentiel linéaire PRNG

Essayons un simple générateur de nombres pseudo-aléatoires, que nous avons déjà considéré une fois dans ce livre (1.29 on page 436) :

```
public class LCG
{
	public static int rand_state;

	public void my_srand (int init)
	{
		rand_state=init;
	}

```

```
        public static int RNG_a=1664525;
        public static int RNG_c=1013904223;

        public int my_rand ()
        {
                rand_state=rand_state*RNG_a;
                rand_state=rand_state+RNG_c;
                return rand_state & 0x7fff;
        }
}
```

Il y a un couple de champs de classe qui sont initialisés au début.

Mais comment? Dans la sortie de `javap`, nous pouvons trouver le constructeur de la classe:

```
  static {};
    flags: ACC_STATIC
    Code:
      stack=1, locals=0, args_size=0
         0: ldc           #5          // int 1664525
         2: putstatic     #3          // Field RNG_a:I
         5: ldc           #6          // int 1013904223
         7: putstatic     #4          // Field RNG_c:I
        10: return
```

C'est ainsi que les variables sont initialisées.

RNG_a occupe le 3ème slot dans la classe et RNG_c—le 4ème, et `putstatic` met les constantes ici.

La fonction `my_srand()` stocke simplement la valeur en entrée dans `rand_state` :

```
  public void my_srand(int);
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=2, args_size=2
         0: iload_1
         1: putstatic     #2          // Field rand_state:I
         4: return
```

`iload_1` prend la valeur en entrée et la pousse sur la pile. Mais pourquoi pas `iload_0` ?

C'est parce que cette fonction peut utiliser les champs de la classe, et donc *this* est aussi passé à la fonction comme paramètre d'indice zéro.

Le champ `rand_state` occupe le 2ème slot dans la classe, donc `putstatic` copie la valeur depuis le TOS dans le 2ème slot.

Maintenant `my_rand()` :

```
  public int my_rand();
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
```

```
         0: getstatic     #2          // Field rand_state:I
         3: getstatic     #3          // Field RNG_a:I
         6: imul
         7: putstatic     #2          // Field rand_state:I
        10: getstatic     #2          // Field rand_state:I
        13: getstatic     #4          // Field RNG_c:I
        16: iadd
        17: putstatic     #2          // Field rand_state:I
        20: getstatic     #2          // Field rand_state:I
        23: sipush        32767
        26: iand
        27: ireturn
```

Ça charge toutes les valeurs depuis les champs de l'objet, effectue l'opération et met à jour les valeurs de `rand_state` en utilisant l'instruction `putstatic`.

À l'offset 20, `rand_state` est rechargé à nouveau (car il a été supprimé de la pile avant, par `putstatic`).

Ceci semble non-efficient, mais soyez assuré que la JVM est en général assez bonne pour vraiment bien optimiser de telles choses.

4.1.8 Conditional jumps

Maintenant, continuons avec les sauts conditionnels.

```
public class abs
{
        public static int abs(int a)
        {
                if (a<0)
                        return -a;
                return a;
        }
}
```

```
  public static int abs(int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iload_0
         1: ifge          7
         4: iload_0
         5: ineg
         6: ireturn
         7: iload_0
         8: ireturn
```

`ifge` saute à l'offset 7 si la valeur du TOS est plus grande ou égale à 0.

N'oubliez pas que chaque instruction `ifXX` supprime la valeur (qui doit être comparée) de la pile.

ineg inverse le signe de la valeur du TOS.

Un autre exemple:

```
        public static int min (int a, int b)
        {
                if (a>b)
                        return b;
                return a;
        }
```

Nous obtenons:

```
  public static int min(int, int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=2
         0: iload_0
         1: iload_1
         2: if_icmple     7
         5: iload_1
         6: ireturn
         7: iload_0
         8: ireturn
```

if_icmple prend deux valeurs et les compare. Si la seconde est plus petite ou égale à la première, un saut à l'offset 7 est effectué.

Quand nous définissons la fonction max() …

```
        public static int max (int a, int b)
        {
                if (a>b)
                        return a;
                return b;
        }
```

…le code résultant est le même, mais les deux dernières instructions iload (aux offsets 5 et 7) sont échangées:

```
  public static int max(int, int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=2
         0: iload_0
         1: iload_1
         2: if_icmple     7
         5: iload_0
         6: ireturn
         7: iload_1
         8: ireturn
```

Un exemple plus avancé:

```
public class cond
{
        public static void f(int i)
        {
                if (i<100)
                        System.out.print("<100");
                if (i==100)
                        System.out.print("==100");
                if (i>100)
                        System.out.print(">100");
                if (i==0)
                        System.out.print("==0");
        }
}
```

```
  public static void f(int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: iload_0
         1: bipush        100
         3: if_icmpge     14
         6: getstatic     #2        // Field java/lang/System.out:Ljava/io/↙
    ↳ PrintStream;
         9: ldc           #3        // String <100
        11: invokevirtual #4        // Method java/io/PrintStream.print:(↙
    ↳ Ljava/lang/String;)V
        14: iload_0
        15: bipush        100
        17: if_icmpne     28
        20: getstatic     #2        // Field java/lang/System.out:Ljava/io/↙
    ↳ PrintStream;
        23: ldc           #5        // String ==100
        25: invokevirtual #4        // Method java/io/PrintStream.print:(↙
    ↳ Ljava/lang/String;)V
        28: iload_0
        29: bipush        100
        31: if_icmple     42
        34: getstatic     #2        // Field java/lang/System.out:Ljava/io/↙
    ↳ PrintStream;
        37: ldc           #6        // String >100
        39: invokevirtual #4        // Method java/io/PrintStream.print:(↙
    ↳ Ljava/lang/String;)V
        42: iload_0
        43: ifne          54
        46: getstatic     #2        // Field java/lang/System.out:Ljava/io/↙
    ↳ PrintStream;
        49: ldc           #7        // String ==0
        51: invokevirtual #4        // Method java/io/PrintStream.print:(↙
    ↳ Ljava/lang/String;)V
        54: return
```

if_icmpge prend deux valeurs et les compare. Si le seconde est plus grande que la première, un saut à l'offset 14 est effectué.

if_icmpne et if_icmple fonctionnent de la même façon, mais implémentent des conditions différentes.

Il y a aussi une instruction ifne à l'offset 43.

Le nom est un terme inapproprié, il aurait été meilleur de l'appeler ifnz (saut si la valeur du TOS n'est pas zéro).

Et c'est ce qu'elle fait: elle saute à l'offset 54 si la valeur en entrée n'est pas zéro.

Si c'est zéro, le flux d'exécution continue à l'offset 46, où la chaîne «==0» est affichée.

N.B.: la JVM n'a pas de type de données non signée, donc les instructions de comparaison opèrent seulement sur des valeurs entières signées.

4.1.9 Passer des paramètres

Étendons notre exemple de min()/max() :

```
public class minmax
{
        public static int min (int a, int b)
        {
                if (a>b)
                        return b;
                return a;
        }

        public static int max (int a, int b)
        {
                if (a>b)
                        return a;
                return b;
        }

        public static void main(String[] args)
        {
                int a=123, b=456;
                int max_value=max(a, b);
                int min_value=min(a, b);
                System.out.println(min_value);
                System.out.println(max_value);
        }
}
```

Voici le code de la fonction main() :

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=5, args_size=1
```

```
         0: bipush        123
         2: istore_1
         3: sipush        456
         6: istore_2
         7: iload_1
         8: iload_2
         9: invokestatic  #2         // Method max:(II)I
        12: istore_3
        13: iload_1
        14: iload_2
        15: invokestatic  #3         // Method min:(II)I
        18: istore        4
        20: getstatic     #4         // Field java/lang/System.out:Ljava/io/⤦
    ↳ PrintStream;
        23: iload         4
        25: invokevirtual #5         // Method java/io/PrintStream.println:(I⤦
    ↳ )V
        28: getstatic     #4         // Field java/lang/System.out:Ljava/io/⤦
    ↳ PrintStream;
        31: iload_3
        32: invokevirtual #5         // Method java/io/PrintStream.println:(I⤦
    ↳ )V
        35: return
```

Les paramètres sont passés à l'autre fonction dans la pile, et la valeur renvoyée est laissée sur le TOS.

4.1.10 Champs de bit

Toutes les opérations au niveau des bits fonctionnent comme sur les autres ISA :

```
        public static int set (int a, int b)
        {
                return a | 1<<b;
        }

        public static int clear (int a, int b)
        {
                return a & (~(1<<b));
        }
```

```
  public static int set(int, int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=2, args_size=2
         0: iload_0
         1: iconst_1
         2: iload_1
         3: ishl
         4: ior
         5: ireturn
```

```
  public static int clear(int, int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=2, args_size=2
         0: iload_0
         1: iconst_1
         2: iload_1
         3: ishl
         4: iconst_m1
         5: ixor
         6: iand
         7: ireturn
```

`iconst_m1` charge -1 sur la pile, c'est la même chose que le nombre `0xFFFFFFFF`.

XORé avec `0xFFFFFFFF` a le même effet qu'inverser tous les bits.

Étendons tous les types de données à 64-bit *long* :

```
        public static long lset (long a, int b)
        {
                return a | 1<<b;
        }

        public static long lclear (long a, int b)
        {
                return a & (~(1<<b));
        }
```

```
  public static long lset(long, int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=3, args_size=2
         0: lload_0
         1: iconst_1
         2: iload_2
         3: ishl
         4: i2l
         5: lor
         6: lreturn

  public static long lclear(long, int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=3, args_size=2
         0: lload_0
         1: iconst_1
         2: iload_2
         3: ishl
         4: iconst_m1
         5: ixor
         6: i2l
         7: land
         8: lreturn
```

Le code est le même, mais des instructions avec le préfixe *l* sont utilisées, qui opèrent avec des valeurs 64-bit.

Ainsi, le second paramètre de la fonction est toujours du type *int*, et lorsque la valeur 32-bit qu'il contient doit être étendues à une valeur 64-bit, l'instruction `i2l` est utilisée,

4.1.11 Boucles

```
public class Loop
{
        public static void main(String[] args)
        {
                for (int i = 1; i <= 10; i++)
                {
                        System.out.println(i);
                }
        }
}
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=1
         0: iconst_1
         1: istore_1
         2: iload_1
         3: bipush        10
         5: if_icmpgt     21
         8: getstatic     #2         // Field java/lang/System.out:Ljava/io↙
    ↳ /PrintStream;
        11: iload_1
        12: invokevirtual #3         // Method java/io/PrintStream.println↙
    ↳ :(I)V
        15: iinc          1, 1
        18: goto          2
        21: return
```

`iconst_1` loads 1 into TOS, `istore_1` stores it in the LVA at slot 1.

Pourquoi pas le slot d'indice zéro? Car la fonction `main()` a un paramètre (tableau de `String`) et un pointeur sur ce dernier (ou une *référence*) qui est maintenant dans le slot 0.

Donc, la variable locale *i* sera toujours dans le premier slot.

Les instructions aux offsets 3 et 5 comparent *i* avec 10.

Si *i* est plus grand, le flux d'exécution passe à l'offset 21, où la fonction se termine.

Si non, `println` est appelée.

i est ensuite rechargé à l'offset 11, pour `println`.

À propos, nous appelons la méthode `println` pour un *entier*, et nous voyons this dans le commentaire: «(I)V» ((*I* signifie *integer* et *V* signifie que le type de retour est *void*).

Lorsque `println` termine, *i* est incrémenté à l'offset 15.

Le premier opérande de l'instruction est le numéro d'un slot (1), le second est le nombre a ajouté à la variable.

Procédons avec un exemple plus complexe:

```
public class Fibonacci
{
        public static void main(String[] args)
        {
                int limit = 20, f = 0, g = 1;

                for (int i = 1; i <= limit; i++)
                {
                        f = f + g;
                        g = f - g;
                        System.out.println(f);
                }
        }
}
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=5, args_size=1
         0: bipush        20
         2: istore_1
         3: iconst_0
         4: istore_2
         5: iconst_1
         6: istore_3
         7: iconst_1
         8: istore        4
        10: iload         4
        12: iload_1
        13: if_icmpgt     37
        16: iload_2
        17: iload_3
        18: iadd
        19: istore_2
        20: iload_2
        21: iload_3
        22: isub
        23: istore_3
        24: getstatic     #2         // Field java/lang/System.out:Ljava/io⤦
    ↳ /PrintStream;
        27: iload_2
        28: invokevirtual #3         // Method java/io/PrintStream.println⤦
    ↳ :(I)V
```

```
        31: iinc          4, 1
        34: goto          10
        37: return
```

Voici une carte des slots LVA :

- 0 — le seul paramètre de `main()`
- 1 — *limit*, contient toujours 20
- 2 — *f*
- 3 — *g*
- 4 — *i*

Nous voyons que le compilateur Java alloue les variables dans des slots LVA dans le même ordre qu'elles sont déclarées dans le code source.

Il y a des instructions `istore` séparées pour accéder aux slots 0, 1, 2 et 3, mais pas pour 4 et plus, donc il y a un `istore` avec un paramètre supplémentaire à l'offset 8 qui prend le numéro du slot comme opérande.

Il y a la même chose avec `iload` à l'offset 10.

Mais n'est-il pas douteux d'allouer un autre slot pour la variable *limit*, qui contient toujours 20 (donc c'est par essence une constante), et de recharger sa valeur si souvent?

Le compilateur JIT de la JVM est en général assez bon pour optimiser de telles choses.

Une intervention manuelle dans le code n'en vaut probablement pas la peine.

4.1.12 switch()

La déclaration switch() est implémentée avec l'instruction `tableswitch` :

```
        public static void f(int a)
        {
                switch (a)
                {
                case 0: System.out.println("zero"); break;
                case 1: System.out.println("one\n"); break;
                case 2: System.out.println("two\n"); break;
                case 3: System.out.println("three\n"); break;
                case 4: System.out.println("four\n"); break;
                default: System.out.println("something unknown\n"); break;
                };
        }
```

Aussi simple que possible:

```
  public static void f(int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
```

```
         0: iload_0
         1: tableswitch   { // 0 to 4
                       0: 36
                       1: 47
                       2: 58
                       3: 69
                       4: 80
                 default: 91
            }
        36: getstatic     #2     // Field java/lang/System.out:Ljava/io/↙
    ↘ PrintStream;
        39: ldc           #3     // String zero
        41: invokevirtual #4     // Method java/io/PrintStream.println:(↙
    ↘ Ljava/lang/String;)V
        44: goto          99
        47: getstatic     #2     // Field java/lang/System.out:Ljava/io/↙
    ↘ PrintStream;
        50: ldc           #5     // String one\n
        52: invokevirtual #4     // Method java/io/PrintStream.println:(↙
    ↘ Ljava/lang/String;)V
        55: goto          99
        58: getstatic     #2     // Field java/lang/System.out:Ljava/io/↙
    ↘ PrintStream;
        61: ldc           #6     // String two\n
        63: invokevirtual #4     // Method java/io/PrintStream.println:(↙
    ↘ Ljava/lang/String;)V
        66: goto          99
        69: getstatic     #2     // Field java/lang/System.out:Ljava/io/↙
    ↘ PrintStream;
        72: ldc           #7     // String three\n
        74: invokevirtual #4     // Method java/io/PrintStream.println:(↙
    ↘ Ljava/lang/String;)V
        77: goto          99
        80: getstatic     #2     // Field java/lang/System.out:Ljava/io/↙
    ↘ PrintStream;
        83: ldc           #8     // String four\n
        85: invokevirtual #4     // Method java/io/PrintStream.println:(↙
    ↘ Ljava/lang/String;)V
        88: goto          99
        91: getstatic     #2     // Field java/lang/System.out:Ljava/io/↙
    ↘ PrintStream;
        94: ldc           #9     // String something unknown\n
        96: invokevirtual #4     // Method java/io/PrintStream.println:(↙
    ↘ Ljava/lang/String;)V
        99: return
```

4.1.13 Tableaux

Exemple simple

Créons d'abord un tableau de 10 entiers et remplissons le:

```
        public static void main(String[] args)
```

```
        {
                int a[]=new int[10];
                for (int i=0; i<10; i++)
                        a[i]=i;
                dump (a);
        }
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=3, args_size=1
         0: bipush        10
         2: newarray       int
         4: astore_1
         5: iconst_0
         6: istore_2
         7: iload_2
         8: bipush        10
        10: if_icmpge     23
        13: aload_1
        14: iload_2
        15: iload_2
        16: iastore
        17: iinc          2, 1
        20: goto          7
        23: aload_1
        24: invokestatic  #4      // Method dump:([I)V
        27: return
```

L'instruction newarray créée un objet tableau de 10 éléments de type *int*.

La taille du tableau est définie par bipush et laissée sur le TOS.

Le type du tableau est mis dans l'opérande de l'instruction newarray.

Après l'exécution de newarray, une *référence* (ou pointeur) sur le tableau nouvellement créé dans le heap est laissée sur le TOS.

astore_1 stocke la *référence* dans le 1er slot dans LVA.

La seconde partie de la fonction main() est la boucle qui stocke *i* dans l'élément du tableau correspondant.

aload_1 obtient une *référence* du tableau et la met sur la pile.

iastore stocke ensuite la valeur entière de la pile dans le tableau, dont la *référence* se trouve dans TOS.

La troisième partie de la fonction main() appelle la fonction dump().

Un argument lui est préparé par aload_1 (offset 23).

Maintenant regardons la fonction dump() :

```
        public static void dump(int a[])
        {
```

```
                for (int i=0; i<a.length; i++)
                        System.out.println(a[i]);
        }
```

```
  public static void dump(int[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=2, args_size=1
         0: iconst_0
         1: istore_1
         2: iload_1
         3: aload_0
         4: arraylength
         5: if_icmpge     23
         8: getstatic     #2      // Field java/lang/System.out:Ljava/io/↙
    ↳ PrintStream;
        11: aload_0
        12: iload_1
        13: iaload
        14: invokevirtual #3      // Method java/io/PrintStream.println:(I)↙
    ↳ V
        17: iinc          1, 1
        20: goto          2
        23: return
```

La `référence` entrante sur le tableau est dans le slot d'indice 0.

L'expression `a.length` dans le code source est convertie en une instruction `arraylength` : elle prend une `référence` sur le tableau et laisse sa taille sur le TOS.

`iaload` à l'offset 13 est utilisée pour charger des éléments du tableau, elle nécessite qu'une *référence* sur le tableau soit présente dans la pile (préparée par `aload_0` en 11), et aussi un index (préparé par `iload_1` à l'offset 12).

Inutile de dire que les instructions préfixées par *a* peuvent être, par erreur, mal interprétées comme des instructions d'*array* (tableaux).

C'est incorrect. Ces instructions travaillent avec des *références* sur les objets.

Et les tableaux et les chaînes sont aussi des objets.

Sommer les éléments d'un tableau

Un autre exemple:

```
public class ArraySum
{
        public static int f (int[] a)
        {
                int sum=0;
                for (int i=0; i<a.length; i++)
                        sum=sum+a[i];
                return sum;
        }
```

```
}
```

```
  public static int f(int[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=3, args_size=1
         0: iconst_0
         1: istore_1
         2: iconst_0
         3: istore_2
         4: iload_2
         5: aload_0
         6: arraylength
         7: if_icmpge     22
        10: iload_1
        11: aload_0
        12: iload_2
        13: iaload
        14: iadd
        15: istore_1
        16: iinc          2, 1
        19: goto          4
        22: iload_1
        23: ireturn
```

Le slot 0 du LVA contient une *référence* sur le tableau en entrée.

Le slot 1 du LVA contient la variable locale *sum*.

Le seul argument de la fonction `main()` est aussi un tableau

Nous allons utiliser le seul argument de la fonction `main()`, qui est un tableau de chaînes:

```
public class UseArgument
{
        public static void main(String[] args)
        {
                System.out.print("Hi, ");
                System.out.print(args[1]);
                System.out.println(". How are you?");
        }
}
```

L'argument d'indice zéro est le nom du programme (comme en C/C++, etc.), donc le 1er argument fourni par l'utilisateur est à l'indice 1.

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=1, args_size=1
         0: getstatic     #2      // Field java/lang/System.out:Ljava/io/↙
    ↘ PrintStream;
```

```
         3: ldc           #3      // String Hi,
         5: invokevirtual #4      // Method java/io/PrintStream.print:(↙
    ↘ Ljava/lang/String;)V
         8: getstatic     #2      // Field java/lang/System.out:Ljava/io/↙
    ↘ PrintStream;
        11: aload_0
        12: iconst_1
        13: aaload
        14: invokevirtual #4      // Method java/io/PrintStream.print:(↙
    ↘ Ljava/lang/String;)V
        17: getstatic     #2      // Field java/lang/System.out:Ljava/io/↙
    ↘ PrintStream;
        20: ldc           #5      // String . How are you?
        22: invokevirtual #6      // Method java/io/PrintStream.println:(↙
    ↘ Ljava/lang/String;)V
        25: return
```

`aload_0` en 11 charge une *référence* sur le slot zéro du LVA (1er et unique argument de `main()`).

`iconst_1` et `aaload` en 12 et 13 prend une *référence* sur l'élément 1 du tableau (en comptant depuis 0).

La *référence* sur l'objet chaîne est sur le TOS à l'offset 14, et elle est prise d'ici par la méthode `println`.

Tableau de chaînes pré-initialisé

```
class Month
{
        public static String[] months =
        {
                "January",
                "February",
                "March",
                "April",
                "May",
                "June",
                "July",
                "August",
                "September",
                "October",
                "November",
                "December"
        };

        public String get_month (int i)
        {
                return months[i];
        };
}
```

La fonction `get_month()` est simple:

```
  public java.lang.String get_month(int);
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=2, args_size=2
         0: getstatic     #2          // Field months:[Ljava/lang/String;
         3: iload_1
         4: aaload
         5: areturn
```

aaload opère sur un tableau de *références*.

Les String Java sont des objets, donc les instructions *a* sont utilisées pour opérer dessus.

areturn renvoie une *référence* sur un objet String.

Comment est initialisé le tableau months[] ?

```
  static {};
    flags: ACC_STATIC
    Code:
      stack=4, locals=0, args_size=0
         0: bipush        12
         2: anewarray     #3          // class java/lang/String
         5: dup
         6: iconst_0
         7: ldc           #4          // String January
         9: aastore
        10: dup
        11: iconst_1
        12: ldc           #5          // String February
        14: aastore
        15: dup
        16: iconst_2
        17: ldc           #6          // String March
        19: aastore
        20: dup
        21: iconst_3
        22: ldc           #7          // String April
        24: aastore
        25: dup
        26: iconst_4
        27: ldc           #8          // String May
        29: aastore
        30: dup
        31: iconst_5
        32: ldc           #9          // String June
        34: aastore
        35: dup
        36: bipush        6
        38: ldc           #10         // String July
        40: aastore
        41: dup
        42: bipush        7
```

```
        44: ldc           #11        // String August
        46: aastore
        47: dup
        48: bipush        8
        50: ldc           #12        // String September
        52: aastore
        53: dup
        54: bipush        9
        56: ldc           #13        // String October
        58: aastore
        59: dup
        60: bipush        10
        62: ldc           #14        // String November
        64: aastore
        65: dup
        66: bipush        11
        68: ldc           #15        // String December
        70: aastore
        71: putstatic     #2         // Field months:[Ljava/lang/String;
        74: return
```

anewarray crée un nouveau tableau de *références* (d'où le préfixe *a*).

Le type de l'objet est défini dans l'opérande de anewarray, dans la chaîne «java/-lang/String».

L'instruction bipush 12 avant anewarray défini la taille du tableau.

Nous voyons une instruction nouvelle pour nous ici: dup.

C'est une instruction standard dans les ordinateurs à pile (langage de programmation Forth inclus) qui duplique simplement la valeur du TOS.

À propos, le FPU 80x87 est aussi un ordinateur à pile et possède une instruction similaire – FDUP.

Elle est utilisée ici pour dupliquer la *référence* sur un tableau, car l'instruction aastore supprime de la pile la *référence* sur le tableau, mais le aastore en aura à nouveau besoin.

Le compilateur Java conclu qui est meilleur de générer un dup plutôt que de générer une instruction getstatic avant chaque opération de stockage (i.e., 11 fois).

aastore pousse une *référence* (sur la chaîne) dans le tableau à un index qui est pris du TOS.

Finalement, putstatic met une *référence* sur le tableau nouvellement créé dans le second champ de notre objet, i.e., le champ *months*.

Fonctions variadiques

Les fonctions variadiques utilisent en fait des tableaux:

```
        public static void f(int... values)
        {
```

```
                for (int i=0; i<values.length; i++)
                        System.out.println(values[i]);
        }

        public static void main(String[] args)
        {
                f (1,2,3,4,5);
        }
```

```
  public static void f(int...);
    flags: ACC_PUBLIC, ACC_STATIC, ACC_VARARGS
    Code:
      stack=3, locals=2, args_size=1
         0: iconst_0
         1: istore_1
         2: iload_1
         3: aload_0
         4: arraylength
         5: if_icmpge     23
         8: getstatic     #2       // Field java/lang/System.out:Ljava/io/↙
    ↳ PrintStream;
        11: aload_0
        12: iload_1
        13: iaload
        14: invokevirtual #3       // Method java/io/PrintStream.println:(I↙
    ↳ )V
        17: iinc          1, 1
        20: goto          2
        23: return
```

`f()` prend juste un tableau d'entier en utilisant `aload_0` à l'offset 3.

Puis, il prend la taille du tableau, etc.

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=1, args_size=1
         0: iconst_5
         1: newarray       int
         3: dup
         4: iconst_0
         5: iconst_1
         6: iastore
         7: dup
         8: iconst_1
         9: iconst_2
        10: iastore
        11: dup
        12: iconst_2
        13: iconst_3
        14: iastore
        15: dup
        16: iconst_3
```

```
        17: iconst_4
        18: iastore
        19: dup
        20: iconst_4
        21: iconst_5
        22: iastore
        23: invokestatic  #4       // Method f:([I)V
        26: return
```

Le tableau est construit dans `main()` en utilisant l'instruction `newarray`, puis il est rempli, et `f()` est appelée.

Oh, à propos, l'objet tableau n'est pas détruit à la fin de `main()`.

Il n'y a pas du tout de destructeurs en Java, car la JVM a un ramasse miette qui fait ceci automatiquement, lorsqu'il sent qu'il doit.

Que dire de la méthode `format()` ?

Elle prend deux arguments en entrée: une chaîne et un tableau d'objets:

```
        public PrintStream format(String format, Object... args)
```

(http://docs.oracle.com/javase/tutorial/java/data/numberformat.html)

Voyons:

```
        public static void main(String[] args)
        {
                int i=123;
                double d=123.456;
                System.out.format("int: %d double: %f.%n", i, d);
        }
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=7, locals=4, args_size=1
         0: bipush        123
         2: istore_1
         3: ldc2_w        #2         // double 123.456d
         6: dstore_2
         7: getstatic     #4         // Field java/lang/System.out:Ljava/io↙
    ↘ /PrintStream;
        10: ldc           #5         // String int: %d double: %f.%n
        12: iconst_2
        13: anewarray     #6         // class java/lang/Object
        16: dup
        17: iconst_0
        18: iload_1
        19: invokestatic  #7         // Method java/lang/Integer.valueOf:(I↙
    ↘ )Ljava/lang/Integer;
        22: aastore
        23: dup
        24: iconst_1
```

```
        25: dload_2
        26: invokestatic  #8         // Method java/lang/Double.valueOf:(D)↙
    ↪ Ljava/lang/Double;
        29: aastore
        30: invokevirtual #9         // Method java/io/PrintStream.format:(↙
    ↪ Ljava/lang/String;[Ljava/lang/Object;)Ljava/io/PrintStream;
        33: pop
        34: return
```

Donc, les valeurs des types *int* et *double* sont d'abord convertis en objets `Integer` et `Double` en utilisant les méthodes `valueOf`.

La méthode `format()` nécessite un objet de type `Object` en entrée, et comme `Integer` et `Double` sont dérivées de la classe racine `Object`, ils conviennent comme éléments du tableau en entrée.

D'un autre côté, un tableau est toujours homogène, i.e., il ne peut pas contenir d'éléments de types différents, ce qui rend impossible de pousser des valeurs *int* et *double* dedans.

Un tableau d'objets `Object` est créé à l'offset 13, un objet `Integer` est ajouté au tableau à l'offset 22, et un objet `Double` est ajouté au tableau à l'offset 29.

La pénultième instruction pop supprime l'élément du TOS, donc lorsque `return` est exécuté, la pile se retrouve vide (ou balancée).

Tableaux bi-dimensionnels

Les tableaux bidimensionnels en Java sont juste des tableaux unidimensionnel de *références* sur d'autres tableaux uni-dimensionnels.

Créons un tableau bi-dimensionnel:

```
        public static void main(String[] args)
        {
                int[][] a = new int[5][10];
                a[1][2]=3;
        }
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=2, args_size=1
         0: iconst_5
         1: bipush        10
         3: multianewarray #2,  2      // class "[[I"
         7: astore_1
         8: aload_1
         9: iconst_1
        10: aaload
        11: iconst_2
        12: iconst_3
        13: iastore
        14: return
```

Il est créé en utilisant l'instruction `multianewarray` : le type de l'objet et ses dimensions sont passés comme opérandes.

La taille du tableau (10*5) est laissée dans la pile (en utilisant les instructions `iconst_5` et `bipush`).

Une *référence* à la line #1 est chargée à l'offset 10 (`iconst_1` et `aaload`).

La colonne est choisie en utilisant `iconst_2` à l'offset 11.

La valeur à écrire est mise à l'offset 12.

`iastore` en 13 écrit l'élément du tableau.

Comment un élément est-il accédé?

```
	public static int get12 (int[][] in)
	{
		return in[1][2];
	}
```

```
  public static int get12(int[][]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: iconst_1
         2: aaload
         3: iconst_2
         4: iaload
         5: ireturn
```

Un *référence* sur la ligne du tableau est chargée à l'offset 2, la colonne est mise à l'offset 3, puis `iaload` charge l'élément du tableau.

Tableaux tri-dimensionnels

Les tableaux tridimensionnels sont simplement des tableaux unidimensionnels de tableaux de *références* sur des tableaux unidimensionnels de *références* de tableaux unidimensionnels.

```
	public static void main(String[] args)
	{
		int[][][] a = new int[5][10][15];

		a[1][2][3]=4;

		get_elem(a);
	}
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=2, args_size=1
```

```
         0: iconst_5
         1: bipush        10
         3: bipush        15
         5: multianewarray #2,  3     // class "[[[I"
         9: astore_1
        10: aload_1
        11: iconst_1
        12: aaload
        13: iconst_2
        14: aaload
        15: iconst_3
        16: iconst_4
        17: iastore
        18: aload_1
        19: invokestatic  #3          // Method get_elem:([[[I)I
        22: pop
        23: return
```

Maintenant, il faut deux instructions `aaload` pour trouver la bonne *référence* :

```
        public static int get_elem (int[][][] a)
        {
                return a[1][2][3];
        }
```

```
  public static int get_elem(int[][][]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: iconst_1
         2: aaload
         3: iconst_2
         4: aaload
         5: iconst_3
         6: iaload
         7: ireturn
```

Résumé

Est-il possible de faire un débordement de tableau en Java?

Non, car la longueur du tableau est toujours présente dans l'objet tableau, les limites du tableau sont contrôlées, et une exception est levée en cas d'accès hors des limites.

Il n'y a pas de tableaux multi-dimensionnels en Java au sens de C/C++, donc Java n'est pas très bien équipé pour des calculs scientifiques rapides.

4.1.14 Chaînes

Premier exemple

Les chaînes sont des objets et sont construites de la même manière que les autres objets (et tableaux).

```
        public static void main(String[] args)
        {
                System.out.println("What is your name?");
                String input = System.console().readLine();
                System.out.println("Hello, "+input);
        }
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=2, args_size=1
         0: getstatic     #2        // Field java/lang/System.out:Ljava/io/↙
    ↳ PrintStream;
         3: ldc           #3        // String What is your name?
         5: invokevirtual #4        // Method java/io/PrintStream.println:(↙
    ↳ Ljava/lang/String;)V
         8: invokestatic  #5        // Method java/lang/System.console:()↙
    ↳ Ljava/io/Console;
        11: invokevirtual #6        // Method java/io/Console.readLine:()↙
    ↳ Ljava/lang/String;
        14: astore_1
        15: getstatic     #2        // Field java/lang/System.out:Ljava/io/↙
    ↳ PrintStream;
        18: new           #7        // class java/lang/StringBuilder
        21: dup
        22: invokespecial #8        // Method java/lang/StringBuilder."<↙
    ↳ init>":()V
        25: ldc           #9        // String Hello,
        27: invokevirtual #10       // Method java/lang/StringBuilder.↙
    ↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
        30: aload_1
        31: invokevirtual #10       // Method java/lang/StringBuilder.↙
    ↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
        34: invokevirtual #11       // Method java/lang/StringBuilder.↙
    ↳ toString:()Ljava/lang/String;
        37: invokevirtual #4        // Method java/io/PrintStream.println:(↙
    ↳ Ljava/lang/String;)V
        40: return
```

La méthode readLine() est appelée à l'offset 11, une *référence* sur la chaîne (qui est fournie par l'utilisateur) est stockée sur le TOS.

À l'offset 14, la *référence* sur la chaîne est stockée dans le slot 1 du LVA.

La chaîne que l'utilisateur a entré est rechargée à l'offset 30 et concaténée avec la chaîne «Hello, » en utilisant la classe StringBuilder.

La chaîne construite est ensuite affichée en utilisant println à l'offset 37.

Second exemple

Un autre exemple:

```
public class strings
{
        public static char test (String a)
        {
                return a.charAt(3);
        };

        public static String concat (String a, String b)
        {
                return a+b;
        }
}
```

```
  public static char test(java.lang.String);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: iconst_3
         2: invokevirtual #2        // Method java/lang/String.charAt:(I)C
         5: ireturn
```

La concaténation de chaînes est réalisée en utilisant `StringBuilder` :

```
  public static java.lang.String concat(java.lang.String, java.lang.String)↙
    ↳ ;
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=2
         0: new           #3        // class java/lang/StringBuilder
         3: dup
         4: invokespecial #4        // Method java/lang/StringBuilder."<↙
    ↳ init>":()V
         7: aload_0
         8: invokevirtual #5        // Method java/lang/StringBuilder.↙
    ↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
        11: aload_1
        12: invokevirtual #5        // Method java/lang/StringBuilder.↙
    ↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
        15: invokevirtual #6        // Method java/lang/StringBuilder.↙
    ↳ toString:()Ljava/lang/String;
        18: areturn
```

Un autre exemple:

```
        public static void main(String[] args)
        {
                String s="Hello!";
                int n=123;
                System.out.println("s=" + s + " n=" + n);
```

```
        }
```

À nouveau, les chaînes sont construites en utilisant la classe `StringBuilder` et sa méthode append, puis la chaîne construite est passée à `println` :

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=3, args_size=1
         0: ldc           #2        // String Hello!
         2: astore_1
         3: bipush        123
         5: istore_2
         6: getstatic     #3        // Field java/lang/System.out:Ljava/io/⤦
    ↳ PrintStream;
         9: new           #4        // class java/lang/StringBuilder
        12: dup
        13: invokespecial #5        // Method java/lang/StringBuilder."<⤦
    ↳ init>":()V
        16: ldc           #6        // String s=
        18: invokevirtual #7        // Method java/lang/StringBuilder.⤦
    ↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
        21: aload_1
        22: invokevirtual #7        // Method java/lang/StringBuilder.⤦
    ↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
        25: ldc           #8        // String  n=
        27: invokevirtual #7        // Method java/lang/StringBuilder.⤦
    ↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
        30: iload_2
        31: invokevirtual #9        // Method java/lang/StringBuilder.⤦
    ↳ append:(I)Ljava/lang/StringBuilder;
        34: invokevirtual #10       // Method java/lang/StringBuilder.⤦
    ↳ toString:()Ljava/lang/String;
        37: invokevirtual #11       // Method java/io/PrintStream.println:(⤦
    ↳ Ljava/lang/String;)V
        40: return
```

4.1.15 Exceptions

Retravaillons un peu notre exemple *Month* (4.1.13 on page 883) :

Listing 4.10 : IncorrectMonthException.java

```
public class IncorrectMonthException extends Exception
{
        private int index;

        public IncorrectMonthException(int index)
        {
                this.index = index;
        }
        public int getIndex()
        {
```

```
                return index;
        }
}
```

Listing 4.11 : Month2.java

```
class Month2
{
        public static String[] months =
        {
                "January",
                "February",
                "March",
                "April",
                "May",
                "June",
                "July",
                "August",
                "September",
                "October",
                "November",
                "December"
        };

        public static String get_month (int i) throws ↙
    ↳ IncorrectMonthException
        {
                if (i<0 || i>11)
                        throw new IncorrectMonthException(i);
                return months[i];
        };

        public static void main (String[] args)
        {
                try
                {
                        System.out.println(get_month(100));
                }
                catch(IncorrectMonthException e)
                {
                        System.out.println("incorrect month index: "+ e.↙
    ↳ getIndex());
                        e.printStackTrace();
                }
        };
}
```

En gros, `IncorrectMonthException.class` possède juste un objet constructeur et une méthode getter.

La classe `IncorrectMonthException` est dérivée d'`Exception`, donc le constructeur de
`IncorrectMonthException` appelle d'abord le constructeur de la classe `Exception`,

puis il met la valeur entière en entrée dans l'unique champ de la classe `IncorrectMonthException` :

```
public IncorrectMonthException(int);
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=2, args_size=2
       0: aload_0
       1: invokespecial #1        // Method java/lang/Exception."<init↙
  ↘ >":()V
       4: aload_0
       5: iload_1
       6: putfield      #2        // Field index:I
       9: return
```

`getIndex()` est simplement un getter. Une *référence* sur `IncorrectMonthException` est passée dans le slot zéro du LVA (*this*), `aload_0` le prend, `getfield` charge une valeur entière depuis l'objet, `ireturn` la renvoie.

```
public int getIndex();
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
       0: aload_0
       1: getfield      #2        // Field index:I
       4: ireturn
```

Maintenant, regardons `get_month()` dans `Month2.class` :

Listing 4.12 : Month2.class

```
public static java.lang.String get_month(int) throws ↙
  ↘ IncorrectMonthException;
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=1, args_size=1
       0: iload_0
       1: iflt          10
       4: iload_0
       5: bipush        11
       7: if_icmple     19
      10: new           #2        // class IncorrectMonthException
      13: dup
      14: iload_0
      15: invokespecial #3        // Method IncorrectMonthException."<↙
  ↘ init>":(I)V
      18: athrow
      19: getstatic     #4        // Field months:[Ljava/lang/String;
      22: iload_0
      23: aaload
      24: areturn
```

`iflt` à l'offset 1 est *if less than*.

Dans le cas d'un index invalide, un nouvel objet est créé en utilisant l'instruction `new` à l'offset 10.

Le type de l'objet est passé comme un opérande à l'instruction (qui est `IncorrectMonthException`).

Ensuite, son constructeur est appelé et l'index est passé via le TOS (offset 15).

Lorsque le contrôle du flux se trouve à l'offset 18, l'objet est déjà construit, donc maintenant l'instruction `athrow` prend une *référence* sur l'objet nouvellement construit et indique à la JVM de trouver le gestionnaire d'exception approprié.

L'instruction `athrow` ne renvoie pas le contrôle du flus ici, donc à l'offset 19 il y a un autre bloc de base, non relatif aux exceptions, où nous pouvons aller depuis l'offset 7.

Comment fonctionnent les gestionnaires?

`main()` in `Month2.class` :

Listing 4.13 : Month2.class

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=2, args_size=1
         0: getstatic     #5         // Field java/lang/System.out:Ljava/io/⤦
    ↳ PrintStream;
         3: bipush        100
         5: invokestatic  #6         // Method get_month:(I)Ljava/lang/⤦
    ↳ String;
         8: invokevirtual #7         // Method java/io/PrintStream.println:(⤦
    ↳ Ljava/lang/String;)V
        11: goto          47
        14: astore_1
        15: getstatic     #5         // Field java/lang/System.out:Ljava/io/⤦
    ↳ PrintStream;
        18: new           #8         // class java/lang/StringBuilder
        21: dup
        22: invokespecial #9         // Method java/lang/StringBuilder."<⤦
    ↳ init>":()V
        25: ldc           #10        // String incorrect month index:
        27: invokevirtual #11        // Method java/lang/StringBuilder.⤦
    ↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
        30: aload_1
        31: invokevirtual #12        // Method IncorrectMonthException.⤦
    ↳ getIndex:()I
        34: invokevirtual #13        // Method java/lang/StringBuilder.⤦
    ↳ append:(I)Ljava/lang/StringBuilder;
        37: invokevirtual #14        // Method java/lang/StringBuilder.⤦
    ↳ toString:()Ljava/lang/String;
        40: invokevirtual #7         // Method java/io/PrintStream.println:(⤦
    ↳ Ljava/lang/String;)V
        43: aload_1
        44: invokevirtual #15        // Method IncorrectMonthException.⤦
    ↳ printStackTrace:()V
        47: return
      Exception table:
         from    to  target type
             0    11    14   Class IncorrectMonthException
```

Ici se trouve la table Exception, qui définit que de l'offset 0 à 11 (inclus), une exception
IncorrectMonthException peut se produire, et si cela se produit, le contrôle du flux sera passé à l'offset 14.

En effet, le programme principal se termine à l'offset 11.

À l'offset 14, le gestionnaire commence. Il n'est pas possible d'arriver ici, il n'y a pas de saut conditionnel/inconditionnel à cet endroit.

Mais la JVM transférera le flux d'exécution ici en cas d'exception.

Le tout premier astore_1 (en 14) prend la *référence* en entrée sur l'objet exception et la stocke dans le slot 1 du LVA.

Plus tard, la méthode getIndex() (de cet objet exception) sera appelée à l'offset 31.

La *référence* sur l'objet exception courant est passée juste avant cela (offset 30).

Le reste du code effectue juste de la manipulation de chaîne: d'abord. la valeur entière renvoyée par getIndex() est convertie en chaîne par la méthode toString(), puis est concaténée avec la chaîne de texte «incorrect month index: » (comme nous l'avons vu avant), enfin println() et printStackTrace() sont appelées.

Après la fin de printStackTrace(), l'exception est gérer et nous pouvons continuer avec l'exécution normale.

À l'offset 47 il y a un return qui termine la fonction main(), mais il pourrait y avoir n'importe quel autre code qui serait exécuté comme si aucune exception n'avait été déclenchée.

Voici un exemple de la façon dont IDA montre les intervalles d'exceptions:

Listing 4.14 : tiré d'un fichier .class quelconque trouvé sur mon ordinateur

```
    .catch java/io/FileNotFoundException from met001_335 to met001_360\
 using met001_360
    .catch java/io/FileNotFoundException from met001_185 to met001_214\
 using met001_214
    .catch java/io/FileNotFoundException from met001_181 to met001_192\
 using met001_195
    .catch java/io/FileNotFoundException from met001_155 to met001_176\
 using met001_176
    .catch java/io/FileNotFoundException from met001_83 to met001_129 using
    \
met001_129
    .catch java/io/FileNotFoundException from met001_42 to met001_66 using
    \
met001_69
    .catch java/io/FileNotFoundException from met001_begin to met001_37\
 using met001_37
```

4.1.16 Classes

Classe simple:

Listing 4.15 : test.java

```
public class test
{
        public static int a;
        private static int b;

        public test()
        {
            a=0;
            b=0;
        }
        public static void set_a (int input)
        {
                a=input;
        }
        public static int get_a ()
        {
                return a;
        }
        public static void set_b (int input)
        {
                b=input;
        }
        public static int get_b ()
        {
                return b;
        }
}
```

Le constructeur met simplement les deux champs à zéro:

```
  public test();
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1         // Method java/lang/Object."<init>":()↙
    ↳ V
         4: iconst_0
         5: putstatic     #2         // Field a:I
         8: iconst_0
         9: putstatic     #3         // Field b:I
        12: return
```

Setter de a :

```
  public static void set_a(int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iload_0
         1: putstatic     #2         // Field a:I
         4: return
```

Getter de a :

```
  public static int get_a();
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=0, args_size=0
         0: getstatic     #2          // Field a:I
         3: ireturn
```

Setter de b :

```
  public static void set_b(int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iload_0
         1: putstatic     #3          // Field b:I
         4: return
```

Getter de b :

```
  public static int get_b();
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=0, args_size=0
         0: getstatic     #3          // Field b:I
         3: ireturn
```

Il n'y a aucune différence dans le code qui fonctionne avec des champs publics ou privés.

Mais ce type d'information est présent dans le fichier `.class` et il n'est pas possible d'accéder aux champs privés depuis n'importe où.

Créons un objet et appelons sa méthode:

Listing 4.16 : ex1.java

```
public class ex1
{
        public static void main(String[] args)
        {
                test obj=new test();
                obj.set_a (1234);
                System.out.println(obj.a);
        }
}
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=1
         0: new           #2         // class test
         3: dup
```

```
         4: invokespecial #3        // Method test."<init>":()V
         7: astore_1
         8: aload_1
         9: pop
        10: sipush        1234
        13: invokestatic  #4        // Method test.set_a:(I)V
        16: getstatic     #5        // Field java/lang/System.out:Ljava/io/↙
    ↳ PrintStream;
        19: aload_1
        20: pop
        21: getstatic     #6        // Field test.a:I
        24: invokevirtual #7        // Method java/io/PrintStream.println:(↙
    ↳ I)V
        27: return
```

L'instruction new crée un objet, mais n'appelle pas le constructeur (il est appelé à l'offset 4).

La méthode `set_a()` est appelée à l'offset 16.

Le champ a est accédé en utilisant l'instruction `getstatic` à l'offset 21.

4.1.17 Correction simple

Premier exemple

Procédons avec une simple de tâche de modification de code.

```
public class nag
{
        public static void nag_screen()
        {
                System.out.println("This program is not registered");
        };
        public static void main(String[] args)
        {
                System.out.println("Greetings from the mega-software");
                nag_screen();
        }
}
```

Comment pourrions-nous supprimer l'affichage de la chaîne «This program is not registered» ?

Chargeons le fichier .class dans IDA :

```
                ; Segment type: Pure code
                  .method public static nag_screen()V
                  .limit stack 2
                  .line 4
178 000 002         getstatic java/lang/System.out Ljava/io/PrintStream;  ; CODE XREF: main+8↓P
018 003             ldc "This program is not registered"
182 000 004         invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                  .line 5
177                 return
??? ??? ???+      .end method
??? ??? ???+
???             ; ---------------------------------------------------------------------------


                ; Segment type: Pure code
                  .method public static main([Ljava/lang/String;)V
                  .limit stack 2
                  .limit locals 1
                  .line 8
178 000 002         getstatic java/lang/System.out Ljava/io/PrintStream;
018 005             ldc "Greetings from the mega-software"
182 000 004         invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                  .line 9
184 000 006         invokestatic nag.nag_screen()V
                  .line 10
177                 return
```

Fig. 4.1: IDA

Modifions le premier octet de la fonction à 177 (qui est l'opcode de l'instruction return) :

```
                ; Segment type: Pure code
                  .method public static nag_screen()V
                  .limit stack 2
                  .line 4

                nag_screen:                                 ; CODE XREF: main+8↓P
177                 return
000                 0 ; 0x00
002                 2 ; 0x02
018 003             ldc "This program is not registered"
182 000 004         invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                  .line 5
177                 return
??? ??? ???+      .end method
??? ??? ???+
???             ; ---------------------------------------------------------------------------
```

Fig. 4.2: IDA

Mais ça ne fonctionne pas (JRE 1.7) :

```
Exception in thread "main" java.lang.VerifyError: Expecting a stack map ↙
    ↳ frame
Exception Details:
  Location:
```

```
    nag.nag_screen()V @1: nop
  Reason:
    Error exists in the bytecode
  Bytecode:
    0000000: b100 0212 03b6 0004 b1

        at java.lang.Class.getDeclaredMethods0(Native Method)
        at java.lang.Class.privateGetDeclaredMethods(Class.java:2615)
        at java.lang.Class.getMethod0(Class.java:2856)
        at java.lang.Class.getMethod(Class.java:1668)
        at sun.launcher.LauncherHelper.getMainMethod(LauncherHelper.java↙
    ↘ :494)
        at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java↙
    ↘ :486)
```

Peut-être que la JVM a d'autres tests relatifs à l'état de la pile.

OK, modifions différemment en supprimant l'appel à `nag()` :

```
                ; Segment type: Pure code
                  .method public static main([Ljava/lang/String;)V
                  .limit stack 2
                  .limit locals 1
                  .line 8
178 000 002         getstatic java/lang/System.out Ljava/io/PrintStream;
018 005             ldc "Greetings from the mega-software"
182 000 004         invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                  .line 9
000                 nop
000                 nop
000                 nop
                  .line 10
177                 return

                ; ===========================================================================
```

Fig. 4.3: IDA

0 est l' opcode de NOP.

Maintenant, ça fonctionne!

Second exemple

Un autre exemple simple de crackme:

```
public class password
{
        public static void main(String[] args)
        {
                System.out.println("Please enter the password");
                String input = System.console().readLine();
                if (input.equals("secret"))
                        System.out.println("password is correct");
```

```
                else
                        System.out.println("password is not correct");
        }
}
```

Chargeons le dans IDA :

```
                    ; Segment type: Pure code
                      .method public static main([Ljava/lang/String;)V
                      .limit stack 2
                      .limit locals 2
                      .line 3
178 000 002             getstatic java/lang/System.out Ljava/io/PrintStream;
018 003                 ldc "Please enter the password"
182 000 004             invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                      .line 4
184 000 005             invokestatic java/lang/System.console()Ljava/io/Console;
182 000 006             invokevirtual java/io/Console.readLine()Ljava/lang/String;
076                     astore_1 ; met002_slot001
                      .line 5
043                     aload_1 ; met002_slot001
018 007                 ldc "secret"
182 000 008             invokevirtual java/lang/String.equals(Ljava/lang/Object;)Z
153 000 014             ifeq met002_35
                      .line 6
178 000 002             getstatic java/lang/System.out Ljava/io/PrintStream;
018 009                 ldc "password is correct"
182 000 004             invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
167 000 011             goto met002_43
                      .line 8

                    met002_35:                                   ; CODE XREF: main+21↑j
178 000 002             .stack use locals
                          locals Object java/lang/String
                        .end stack
                        getstatic java/lang/System.out Ljava/io/PrintStream;
018 010                 ldc "password is not correct"
182 000 004             invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                      .line 9
```

Fig. 4.4: IDA

Nous voyons ici l'instruction `ifeq` qui effectue le travail.

Son nom signifie *if equal* (si égal), et c'est mal-nommé, un meilleur nom serait `ifz` (*if zero*, si zéro), i.e, si la valeur sur le TOS est zéro, alors effectuer le saut.

Dans notre exemple, le saut est fait si le mot de passe est incorrect (la méthode `equals` renvoie `False`, qui est 0).

La toute première idée est de patcher cette instruction.

Il y a deux octets dans l'opcode de `ifeq`, qui encode l'offset du saut.

Pour transformer cette instruction en NOP, nous devons mettre la valeur 3 dans le 3ème octet (car ajouter 3 à l'adresse courante sautera toujours à l'instruction suivante, puisque la longueur de l'instruction `ifeq` est de 3 octets) :

```
                  ; Segment type: Pure code
                    .method public static main([Ljava/lang/String;)V
                    .limit stack 2
                    .limit locals 2
                    .line 3
178 000 002           getstatic java/lang/System.out Ljava/io/PrintStream;
018 003               ldc "Please enter the password"
182 000 004           invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                    .line 4
184 000 005           invokestatic java/lang/System.console()Ljava/io/Console;
182 000 006           invokevirtual java/io/Console.readLine()Ljava/lang/String;
076                   astore_1 ; met002_slot001
                    .line 5
043                   aload_1 ; met002_slot001
018 007               ldc "secret"
182 000 008           invokevirtual java/lang/String.equals(Ljava/lang/Object;)Z
153 000 003           ifeq met002_24
                    .line 6

                  met002_24:                                   ; CODE XREF: main+21↑j
178 000 002           getstatic java/lang/System.out Ljava/io/PrintStream;
018 009               ldc "password is correct"
182 000 004           invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
167 000 011           goto met002_43
                    .line 8
178 000 002           .stack use locals
                        locals Object java/lang/String
                      .end stack
                      getstatic java/lang/System.out Ljava/io/PrintStream;
018 010               ldc "password is not correct"
182 000 004           invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                    .line 9
```

Fig. 4.5: IDA

Ceci ne fonctionne pas (JRE 1.7) :

```
Exception in thread "main" java.lang.VerifyError: Expecting a stackmap ↙
    ↘ frame at branch target 24
Exception Details:
  Location:
    password.main([Ljava/lang/String;)V @21: ifeq
  Reason:
    Expected stackmap frame at this location.
  Bytecode:
    0000000: b200 0212 03b6 0004 b800 05b6 0006 4c2b
    0000010: 1207 b600 0899 0003 b200 0212 09b6 0004
    0000020: a700 0bb2 0002 120a b600 04b1
  Stackmap Table:
    append_frame(@35,Object[#20])
    same_frame(@43)

        at java.lang.Class.getDeclaredMethods0(Native Method)
        at java.lang.Class.privateGetDeclaredMethods(Class.java:2615)
        at java.lang.Class.getMethod0(Class.java:2856)
        at java.lang.Class.getMethod(Class.java:1668)
        at sun.launcher.LauncherHelper.getMainMethod(LauncherHelper.java↙
```

```
    ↳ :494)
        at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java↙
    ↳ :486)
```

Mais il faut mentionner que ceci fonctionnait avec le JRE 1.6.

Nous pouvons aussi essayer de remplacer les 3 octets de l'opcode de `ifeq` par des octets à zéro (NOP), mais ça ne fonctionne toujours pas.

Il semble qu'il y a plus de tests de l'état de la pile en JRE 1.7.

OK, remplaçons tout l'appel à la méthode `equals` avec l'instruction `iconst_1` et quelques NOPs:

```
                ; Segment type: Pure code
                  .method public static main([Ljava/lang/String;)V
                  .limit stack 2
                  .limit locals 2
                  .line 3
178 000 002         getstatic java/lang/System.out Ljava/io/PrintStream;
018 003             ldc "Please enter the password"
182 000 004         invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                  .line 4
184 000 005         invokestatic java/lang/System.console()Ljava/io/Console;
182 000 006         invokevirtual java/io/Console.readLine()Ljava/lang/String;
076                 astore_1 ; met002_slot001
                  .line 5
004                 iconst_1
000                 nop
000                 nop
000                 nop
000                 nop
000                 nop
153 000 014         ifeq met002_35
                  .line 6
178 000 002         getstatic java/lang/System.out Ljava/io/PrintStream;
018 009             ldc "password is correct"
182 000 004         invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
167 000 011         goto met002_43
                  .line 8

                met002_35:                                  ; CODE XREF: main+21↑j
178 000 002         .stack use locals
                      locals Object java/lang/String
                    .end stack
```

Fig. 4.6: IDA

Il doit toujours y avoir 1 sur le TOS lorsque l'instruction `ifeq` est exécutée, ainsi `ifeq` ne fera jamais le saut.

Ceci fonctionne.

4.1.18 Résumé

Que manque-t-il à Java par rapport à C/C++ ?

- Structures: utiliser les classes.
- Unions; utiliser des hiérarchies de classes.
- Types de données non signés. À propos, ceci rend les algorithmes cryptographiques quelque peu plus difficile à implémenter en Java.
- Pointeurs de fonction.

Chapitre 5

Trouver des choses importantes/intéressantes dans le code

Le minimalisme n'est pas une caractéristique prépondérante des logiciels modernes.

Pas parce que les programmeurs écrivent beaucoup, mais parce que de nombreuses bibliothèques sont couramment liées statiquement aux fichiers exécutable. Si toutes les bibliothèques externes étaient déplacées dans des fichiers DLL externes, le monde serait différent. (Une autre raison pour C++ sont la STL et autres bibliothèques templates.)

Ainsi, il est très important de déterminer l'origine de la fonction, si elle provient d'une bibliothèque standard ou d'une bibliothèque bien connue (comme Boost[1], libpng[2]), ou si elle est liée à ce que l'on essaye de trouver dans le code.

Il est simplement absurde de tout récrire le code en C/C++ pour trouver ce que l'on cherche.

Une des premières tâches d'un rétro-ingénieur est de trouver rapidement le code dont il a besoin.

Le dés-assembleur IDA nous permet de chercher parmi les chaînes de texte, les séquences d'octets et les constantes. Il est même possible d'exporter le code dans un fichier texte .lst ou .asm et d'utiliser `grep`, awk, etc.

Lorsque vous essayez de comprendre ce que fait un certain code, ceci peut être facile avec une bibliothèque open-source comme libpng. Donc, lorsque vous voyez certaines constantes ou chaînes de texte qui vous semblent familières, il vaut toujours la peine de les *googler*. Et si vous trouvez le projet open-source où elles sont utilisées, alors il suffit de comparer les fonctions. Ceci peut permettre de résoudre certaines parties du problème.

[1] http://www.boost.org/
[2] http://www.libpng.org/pub/png/libpng.html

Par exemple, si un programme utilise des fichiers XML, la premières étape peut-être de déterminer quelle bibliothèque XML est utilisée pour le traitement, puisque les bibliothèques standards (ou bien connues) sont en général utilisées au lieu de code fait maison.

Par exemple, j'ai essayé une fois de comprendre comment la compression/décompression des paquets réseau fonctionne dans SAP 6.0. C'est un logiciel gigantesque, mais un .PDB détaillé avec des informations de débogage est présent, et c'est pratique. J'en suis finalement arrivé à l'idée que l'une des fonctions, qui était appelée par *CsDecomprLZC*, effectuait la décompression des paquets réseau. Immédiatement, j'ai essayé de googler le nom et rapidement trouvé que la fonction était utilisée dans MaxDB (c'est un projet open-source de SAP) [3].

`http://www.google.com/search?q=CsDecomprLZC`

Étonnement, les logiciels MaxDB et SAP 6.0 partagent du code comme ceci pour la compression/ décompression des paquets réseau.

5.1 Identification de fichiers exécutables

5.1.1 Microsoft Visual C++

Les versions de MSVC et des DLLs peuvent être importées:

Marketing ver.	Internal ver.	CL.EXE ver.	DLLs imported	Release date
6	6.0	12.00	msvcrt.dll msvcp60.dll	June 1998
.NET (2002)	7.0	13.00	msvcr70.dll msvcp70.dll	February 13, 2002
.NET 2003	7.1	13.10	msvcr71.dll msvcp71.dll	April 24, 2003
2005	8.0	14.00	msvcr80.dll msvcp80.dll	November 7, 2005
2008	9.0	15.00	msvcr90.dll msvcp90.dll	November 19, 2007
2010	10.0	16.00	msvcr100.dll msvcp100.dll	April 12, 2010
2012	11.0	17.00	msvcr110.dll msvcp110.dll	September 12, 2012
2013	12.0	18.00	msvcr120.dll msvcp120.dll	October 17, 2013

msvcp*.dll contient des fonctions relatives à C++, donc si elle est importées, il s'agit probablement d'un programme C++.

Mangling de nom

Les noms commencent en général par le symbole ?.

[3] Plus sur ce sujet dans la section concernée (8.12.1 on page 1141)

Vous trouverez plus d'informations le mangling de nom de MSVC ici: 3.21.1 on page 699.

5.1.2 GCC

À part les cibles *NIX, GCC est aussi présent dans l'environnement win32, sous la forme de Cygwin et MinGW.

Mangling de nom

Les noms commencent en général par le symbole _Z. Vous trouverez plus d'informations le mangling de nom de GCC ici: 3.21.1 on page 699.

Cygwin

cygwin1.dll est souvent importée.

MinGW

msvcrt.dll peut être importée.

5.1.3 Intel Fortran

libifcoremd.dll, libifportmd.dll et libiomp5md.dll (support OpenMP) peuvent être importées.

libifcoremd.dll a beaucoup de fonctions préfixées par `for_`, qui signifie *Fortran*.

5.1.4 Watcom, OpenWatcom

Mangling de nom

Les noms commencent usuellement par le symbole `W`.

Par exemple, ceci est la façon dont la méthode nommées «method» de la classe «class» qui n'a pas d'argument et qui renvoie *void* est encodée:

```
W?method$_class$n__v
```

5.1.5 Borland

Voici un exemple de mangling de nom de Delphi de Borland et de C++Builder:

```
@TApplication@IdleAction$qv
@TApplication@ProcessMDIAccels$qp6tagMSG
@TModule@$bctr$qpcpvt1
@TModule@$bdtr$qv
@TModule@ValidWindow$qp14TWindowsObject
@TrueColorTo8BitN$qpviiiiiit1iiiiii
@TrueColorTo16BitN$qpviiiiiit1iiiiii
```

```
@DIB24BitTo8BitBitmap$qpviiiiiit1iiiii
@TrueBitmap@$bctr$qpcl
@TrueBitmap@$bctr$qpvl
@TrueBitmap@$bctr$qiilll
```

Les noms commencent toujours avec le symbole @, puis nous avons le nom de la classe, de la méthode et les types des arguments de méthode encodés.

Ces noms peuvent être dans des imports .exe, des exports .dll, des données de débogage, etc.

Les Borland Visual Component Libraries (VCL) sont stockées dans des fichiers .bpl au lieu de .dll, par exemple, vcl50.dll, rtl60.dll.

Une autre DLL qui peut être importée: BORLNDMM.DLL.

Delphi

Presque tous les exécutables Delpi ont la chaîne de texte «Boolean» au début de leur segment de code, ainsi que d'autres noms de type.

Ceci est le début très typique du segment CODE d'un programme Delphi, ce bloc vient juste après l'entête de fichier win32 PE:

```
04 10 40 00 03 07 42 6f 6f 6c 65 61 6e 01 00 00  |..@...Boolean...|
00 00 01 00 00 00 00 10 40 00 05 46 61 6c 73 65  |........@..False|
04 54 72 75 65 8d 40 00 2c 10 40 00 09 08 57 69  |.True.@.,.@...Wi|
64 65 43 68 61 72 03 00 00 00 00 ff ff 00 00 90  |deChar..........|
44 10 40 00 02 04 43 68 61 72 01 00 00 00 00 ff  |D.@...Char......|
00 00 00 90 58 10 40 00 01 08 53 6d 61 6c 6c 69  |....X.@...Smalli|
6e 74 02 00 80 ff ff ff 7f 00 00 90 70 10 40 00  |nt..........p.@.|
01 07 49 6e 74 65 67 65 72 04 00 00 00 80 ff ff  |..Integer.......|
ff 7f 8b c0 88 10 40 00 01 04 42 79 74 65 01 00  |......@...Byte..|
00 00 00 ff 00 00 00 90 9c 10 40 00 01 04 57 6f  |..........@...Wo|
72 64 03 00 00 00 00 ff ff 00 00 90 b0 10 40 00  |rd............@.|
01 08 43 61 72 64 69 6e 61 6c 05 00 00 00 00 ff  |..Cardinal......|
ff ff ff 90 c8 10 40 00 10 05 49 6e 74 36 34 00  |......@...Int64.|
00 00 00 00 00 00 80 ff ff ff ff ff ff ff 7f 90  |................|
e4 10 40 00 04 08 45 78 74 65 6e 64 65 64 02 90  |..@...Extended..|
f4 10 40 00 04 06 44 6f 75 62 6c 65 01 8d 40 00  |..@...Double..@.|
04 11 40 00 04 08 43 75 72 72 65 6e 63 79 04 90  |..@...Currency..|
14 11 40 00 0a 06 73 74 72 69 6e 67 20 11 40 00  |..@...string .@.|
0b 0a 57 69 64 65 53 74 72 69 6e 67 30 11 40 00  |..WideString0.@.|
0c 07 56 61 72 69 61 6e 74 8d 40 00 40 11 40 00  |..Variant.@.@.@.|
0c 0a 4f 6c 65 56 61 72 69 61 6e 74 98 11 40 00  |..OleVariant..@.|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 98 11 40 00  |..............@.|
04 00 00 00 00 00 00 00 18 4d 40 00 24 4d 40 00  |.........M@.$M@.|
28 4d 40 00 2c 4d 40 00 20 4d 40 00 68 4a 40 00  |(M@.,M@. M@.hJ@.|
84 4a 40 00 c0 4a 40 00 07 54 4f 62 6a 65 63 74  |.J@..J@..TObject|
a4 11 40 00 07 07 54 4f 62 6a 65 63 74 98 11 40  |..@...TObject..@|
00 00 00 00 00 00 00 06 53 79 73 74 65 6d 00 00  |........System..|
c4 11 40 00 0f 0a 49 49 6e 74 65 72 66 61 63 65  |..@...IInterface|
00 00 00 00 01 00 00 00 00 00 00 00 00 c0 00 00  |................|
```

```
00 00 00 00 46 06 53 79 73 74 65 6d 03 00 ff ff  |....F.System....|
f4 11 40 00 0f 09 49 44 69 73 70 61 74 63 68 c0  |..@...IDispatch.|
11 40 00 01 00 04 02 00 00 00 00 c0 00 00 00     |.@.............|
00 00 00 46 06 53 79 73 74 65 6d 04 00 ff ff 90  |...F.System.....|
cc 83 44 24 04 f8 e9 51 6c 00 00 83 44 24 04 f8  |..D$...Ql...D$..|
e9 6f 6c 00 00 83 44 24 04 f8 e9 79 6c 00 00 cc  |.ol...D$...yl...|
cc 21 12 40 00 2b 12 40 00 35 12 40 00 01 00 00  |.!.@.+.@.5.@....|
00 00 00 00 00 00 00 00 00 c0 00 00 00 00 00 00  |................|
46 41 12 40 00 08 00 00 00 00 00 00 00 8d 40 00  |FA.@..........@.|
bc 12 40 00 4d 12 40 00 00 00 00 00 00 00 00 00  |..@.M.@.........|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
bc 12 40 00 0c 00 00 00 4c 11 40 00 18 4d 40 00  |..@.....L.@..M@.|
50 7e 40 00 5c 7e 40 00 2c 4d 40 00 20 4d 40 00  |P~@.\~@.,M@. M@.|
6c 7e 40 00 84 4a 40 00 c0 4a 40 00 11 54 49 6e  |l~@..J@..J@..TIn|
74 65 72 66 61 63 65 64 4f 62 6a 65 63 74 8b c0  |terfacedObject..|
d4 12 40 00 07 11 54 49 6e 74 65 72 66 61 63 65  |..@...TInterface|
64 4f 62 6a 65 63 74 bc 12 40 00 a0 11 40 00 00  |dObject..@...@..|
00 06 53 79 73 74 65 6d 00 00 8b c0 00 13 40 00  |..System......@.|
11 0b 54 42 6f 75 6e 64 41 72 72 61 79 04 00 00  |..TBoundArray...|
00 00 00 00 00 03 00 00 00 6c 10 40 00 06 53 79  |.........l.@..Sy|
73 74 65 6d 28 13 40 00 04 09 54 44 61 74 65 54  |stem(.@...TDateT|
69 6d 65 01 ff 25 48 e0 c4 00 8b c0 ff 25 44 e0  |ime..%H......%D.|
```

Les 4 premiers octets du segment de données (DATA) peuvent être `00 00 00 00`, `32 13 8B C0` ou `FF FF FF FF`.

Cette information peut être utile lorsque l'on fait face à des exécutables Delphi préparés/chiffrés.

5.1.6 Autres DLLs connues

- vcomp*.dll—implémentation d'OpenMP de Microsoft.

5.2 Communication avec le monde extérieur (niveau fonction)

Il est souvent recommandé de suivre les arguments de la fonction et sa valeur de retour dans un débogueur ou DBI. Par exemple, l'auteur a essayé une fois de comprendre la signification d'une fonction obscure, qui s'est avérée être un tri à bulles mal implémenté[4]. (Il fonctionnait correctement, mais plus lentement.) En même temps, regarder les entrées et sorties de cette fonction aide instantanément à comprendre ce quelle fait.

Souvent, lorsque vous voyez une division par la multiplication (3.12 on page 636), mais avez oublié tous les détails du mécanisme, vous pouvez seulement observer l'entrée et la sortie, et trouver le diviseur rapidement.

[4]https://yurichev.com/blog/weird_sort_KLEE/

5.3 Communication avec le monde extérieur (win32)

Parfois, il est suffisant d'observer les entrées/sorties d'une fonction pour comprendre ce qu'elle fait. Ainsi, vous pouvez gagner du temps.

Accès aux fichiers et au registre: pour les analyses très basiques, l'utilitaire, Process Monitor[5] de SysInternals peut aider.

Pour l'analyse basique des accès au réseau, Wireshark[6] peut être utile.

Mais vous devrez de toutes façons regarder à l'intérieur,

Les premières choses à chercher sont les fonctions des APIs de l'OS et des bibliothèques standards qui sont utilisées.

Si le programme est divisé en un fichier exécutable et un groupe de fichiers DLL, parfois le nom des fonctions dans ces DLLs peut aider.

Si nous sommes intéressés par exactement ce qui peut conduire à appeler `MessageBox()` avec un texte spécifique, nous pouvons essayer de trouver ce texte dans le segment de données, trouver sa référence et trouver les points depuis lesquels le contrôle peut être passé à l'appel à `MessageBox()` qui nous intéresse.

Si nous parlons d'un jeu vidéo et que nous sommes intéressés par les évènements qui y sont plus ou moins aléatoires, nous pouvons essayer de trouver la fonction `rand()` ou sa remplaçante (comme l'algorithme du twister de Mersenne) et trouver les points depuis lesquels ces fonctions sont appelées, et plus important, comment les résultats sont utilisés. Un exemple: 8.3 on page 1044.

Mais si ce n'est pas un jeu, et que `rand()` est toujours utilisé, il est intéressant de savoir pourquoi. Il a y des cas d'utilisation inattendu de `rand()` dans des algorithmes de compression de données (pour une imitation du chiffrement) : blog.yurichev.com.

5.3.1 Fonctions souvent utilisées dans l'API Windows

Ces fonctions peuvent être parmi les fonctions importées. Il est utile de noter que toutes les fonctions ne sont pas forcément utilisées dans du code écrit par le programmeur. Beaucoup de fonctions peuvent être appelées depuis des fonctions de bibliothèque et du code CRT.

Certaines fonctions peuvent avoir le suffixe `-A` pour la version ASCII et `-W` pour la version Unicode.

- Accès au registre (advapi32.dll) : RegEnumKeyEx, RegEnumValue, RegGetValue, RegOpenKeyEx, RegQueryValueEx.
- Accès au text des fichiers .ini (kernel32.dll) : GetPrivateProfileString.
- Boites de dialogue (user32.dll) : MessageBox, MessageBoxEx, CreateDialog, SetDlgItemText, GetDlgItemText.
- Accès aux resources (6.5.2 on page 994) : (user32.dll) : LoadMenu.

[5] http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx
[6] http://www.wireshark.org/

- Réseau TCP/IP (ws2_32.dll) : WSARecv, WSASend.
- Accès fichier (kernel32.dll) : CreateFile, ReadFile, ReadFileEx, WriteFile, WriteFileEx.
- Accès haut niveau à Internet (wininet.dll) : WinHttpOpen.
- Vérifier la signature digitale d'uin fichier exécutable (wintrust.dll) : WinVerifyTrust.
- La bibliothèque MSVC standard (si elle est liée dynamiquement) (msvcr*.dll) : assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr, strchr.

5.3.2 Étendre la période d'essai

Les fonctions d'accès au registre sont des cibles fréquentes pour ceux qui veulent essayer de craquer des logiciels avec période d'essai, qui peuvent sauvegarder la date et l'heure dans un registre.

Des autres cibles courantes sont les fonctions GetLocalTime() et GetSystemTime() : un logiciel avec période d'essai, à chaque démarrage, doit de toutes façons vérifier la date et l'heure d'une certaine façon.

5.3.3 Supprimer la boite de dialogue nag

Une manière répandue de trouver ce qui cause l'apparition de la boite de dialogue nag est d'intercepter les fonctions MessageBox(), CreateDialog() et CreateWindow().

5.3.4 tracer: Intercepter toutes les fonctions dans un module spécifique

Il y a un point d'arrêt INT3 dans tracer, qui peut être déclenché seulement une fois, toutefois, il peut être mis pour toutes les fonctions dans une DLL spécifique.

```
--one-time-INT3-bp:somedll.dll!.*
```

Ou, mettons un point d'arrêt INT3 sur toutes les fonctions avec le préfixe `xml` dans leur nom:

```
--one-time-INT3-bp:somedll.dll!xml.*
```

Le revers de la médaille est que de tels points d'arrêt ne sont déclenchés qu'une fois. Tracer montrera l'appel à une fonction, s'il se produit, mais seulement une fois. Un autre inconvénient—il est impossible de voir les arguments de la fonction.

Néanmoins, cette fonctionnalité est très utile lorsque vous avez qu'un programme utilise une DLL, mais que vous ne savez pas quelles fonctions sont effectivement utilisées. Et il y a beaucoup de fonctions.

Par exemple, regardons ce qu'utilise l'utilitaire uptime de Cygwin:

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

Ainsi nous pouvons voir quelles sont les fonctions de la bibliothèque cygwin1.dll qui sont appelées au moins une fois, et depuis où:

```
One-time INT3 breakpoint: cygwin1.dll!__main (called from uptime.exe!OEP+0↙
    ↳ x6d (0x40106d))
One-time INT3 breakpoint: cygwin1.dll!_geteuid32 (called from uptime.exe!↙
    ↳ OEP+0xba3 (0x401ba3))
One-time INT3 breakpoint: cygwin1.dll!_getuid32 (called from uptime.exe!OEP↙
    ↳ +0xbaa (0x401baa))
One-time INT3 breakpoint: cygwin1.dll!_getegid32 (called from uptime.exe!↙
    ↳ OEP+0xcb7 (0x401cb7))
One-time INT3 breakpoint: cygwin1.dll!_getgid32 (called from uptime.exe!OEP↙
    ↳ +0xcbe (0x401cbe))
One-time INT3 breakpoint: cygwin1.dll!sysconf (called from uptime.exe!OEP+0↙
    ↳ x735 (0x401735))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from uptime.exe!OEP↙
    ↳ +0x7b2 (0x4017b2))
One-time INT3 breakpoint: cygwin1.dll!_open64 (called from uptime.exe!OEP+0↙
    ↳ x994 (0x401994))
One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from uptime.exe!OEP↙
    ↳ +0x7ea (0x4017ea))
One-time INT3 breakpoint: cygwin1.dll!read (called from uptime.exe!OEP+0↙
    ↳ x809 (0x401809))
One-time INT3 breakpoint: cygwin1.dll!sscanf (called from uptime.exe!OEP+0↙
    ↳ x839 (0x401839))
One-time INT3 breakpoint: cygwin1.dll!uname (called from uptime.exe!OEP+0↙
    ↳ x139 (0x401139))
One-time INT3 breakpoint: cygwin1.dll!time (called from uptime.exe!OEP+0↙
    ↳ x22e (0x40122e))
One-time INT3 breakpoint: cygwin1.dll!localtime (called from uptime.exe!OEP↙
    ↳ +0x236 (0x401236))
One-time INT3 breakpoint: cygwin1.dll!sprintf (called from uptime.exe!OEP+0↙
    ↳ x25a (0x40125a))
One-time INT3 breakpoint: cygwin1.dll!setutent (called from uptime.exe!OEP↙
    ↳ +0x3b1 (0x4013b1))
One-time INT3 breakpoint: cygwin1.dll!getutent (called from uptime.exe!OEP↙
    ↳ +0x3c5 (0x4013c5))
One-time INT3 breakpoint: cygwin1.dll!endutent (called from uptime.exe!OEP↙
    ↳ +0x3e6 (0x4013e6))
One-time INT3 breakpoint: cygwin1.dll!puts (called from uptime.exe!OEP+0↙
    ↳ x4c3 (0x4014c3))
```

5.4 Chaînes

5.4.1 Chaînes de texte

C/C++

Les chaînes C normales sont terminées par un zéro (chaînes ASCIIZ).

La raison pour laquelle le format des chaînes C est ce qu'il est (terminé par zéro) est apparemment historique: Dans [Dennis M. Ritchie, *The Evolution of the Unix Time-sharing System*, (1979)] nous lisons:

> A minor difference was that the unit of I/O was the word, not the byte, because the PDP-7 was a word-addressed machine. In practice this meant merely that all programs dealing with character streams ignored null characters, because null was used to pad a file to an even number of characters.

Une différence mineure était que l'unité d'E/S était le mot, pas l'octet, car le PDP-7 était une machine adressée par mot. En pratique, cela signifiait que tous les programmes ayant à faire avec des flux de caractères ignoraient le caractère nul, car nul était utilisé pour compléter un fichier ayant un nombre impair de caractères.

Dans Hiew ou FAR Manager ces chaînes ressemblent à ceci:

```
int main()
{
        printf ("Hello, world!\n");
};
```

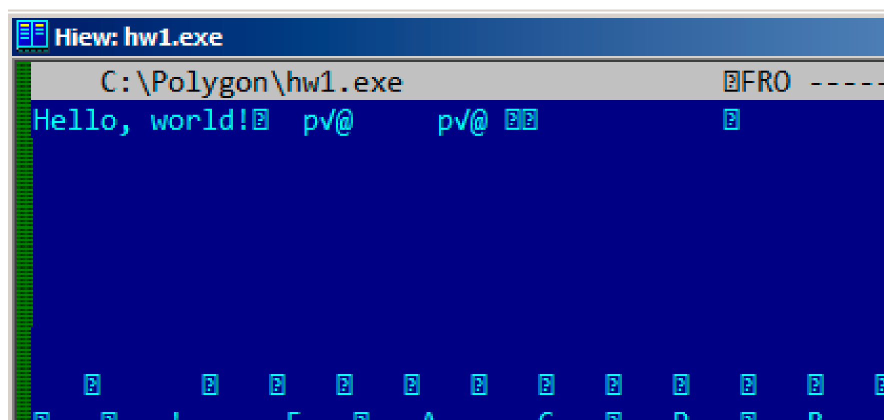


Fig. 5.1: Hiew

Borland Delphi

Une chaîne en Pascal et en Delphi de Borland est précédée par sa longueur sur 8-bit ou 32-bit.

Par exemple:

Listing 5.1 : Delphi

```
CODE:00518AC8                 dd 19h
CODE:00518ACC aLoading___Plea db 'Loading... , please wait.',0
```

```
...

CODE:00518AFC                 dd 10h
CODE:00518B00 aPreparingRun__ db 'Preparing run...',0
```

Unicode

Souvent, ce qui est appelé Unicode est la méthode pour encoder des chaînes où chaque caractère occupe 2 octets ou 16 bits. Ceci est une erreur de terminologie répandue. Unicode est un standard pour assigner un nombre à chaque caractère dans un des nombreux systèmes d'écriture dans le monde, mais ne décrit pas la méthode d'encodage.

Les méthodes d'encodage les plus répandues sont: UTF-8 (est répandue sur Internet et les systèmes *NIX) et UTF-16LE (est utilisé dans Windows).

UTF-8

UTF-8 est l'une des méthodes les plus efficace pour l'encodage des caractères. Tous les symboles Latin sont encodés comme en ASCII, et les symboles après la table ASCII sont encodés en utilisant quelques octets. 0 est encodé comme avant, donc toutes les fonctions C de chaîne standard fonctionnent avec des chaînes UTF-8 comme avec tout autre chaîne.

Voyons comment les symboles de divers langages sont encodés en UTF-8 et de quoi ils ont l'air en FAR, en utilisant la page de code 437[7] :

```
How much? 100€?

(English) I can eat glass and it doesn't hurt me.
(Greek) Μπορώ να φάω σπασμένα γυαλιά χωρίς να πάθω τίποτα.
(Hungarian) Meg tudom enni az üveget, nem lesz tőle bajom.
(Icelandic) Ég get etið gler án þess að meiða mig.
(Polish) Mogę jeść szkło i mi nie szkodzi.
(Russian) Я могу есть стекло, оно мне не вредит.
(Arabic): أنا قادر على أكل الزجاج و هذا لا يؤلمني.
(Hebrew): אני יכול לאכול זכוכית וזה לא מזיק לי.
(Chinese) 我能吞下玻璃而不伤身体。
(Japanese) 私はガラスを食べられます。それは私を傷つけません。
(Hindi) मैं काँच खा सकता हूँ और मुझे उससे कोई चोट नहीं पहुंचती.
```

[7] L'exemple et les traductions ont été pris d'ici: http://www.columbia.edu/~fdc/utf8/

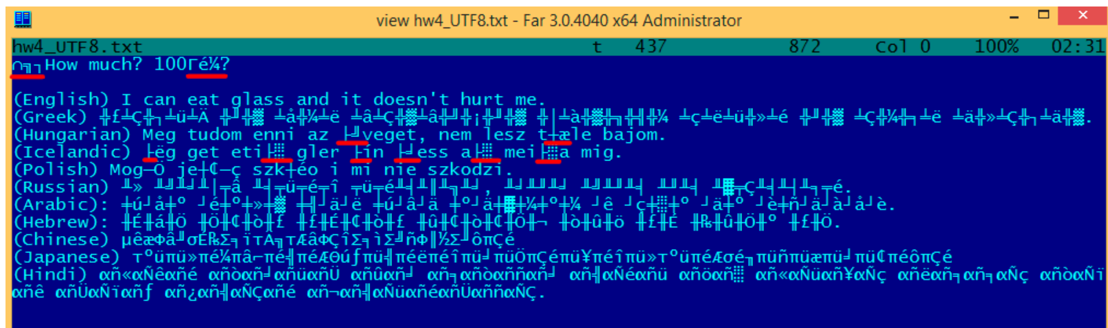


Fig. 5.2: FAR: UTF-8

Comme vous le voyez, la chaîne en anglais est la même qu'en ASCII.

Le hongrois utilise certains symboles Latin et des symboles avec des signes diacritiques.

Ces symboles sont encodés en utilisant plusieurs octets, qui sont soulignés en rouge. C'est le même principe avec l'islandais et le polonais.

Il y a aussi le symbole de l'«Euro» au début, qui est encodé avec 3 octets.

Les autres systèmes d'écritures n'ont de point commun avec Latin.

Au moins en russe, arabe hébreux et hindi, nous pouvons voir des octets récurrents, et ce n'est pas une surprise: tous les symboles d'un système d'écriture sont en général situés dans la même table Unicode, donc leur code débute par le même nombre.

Au début, avant la chaîne «How much?», nous voyons 3 octets, qui sont en fait le BOM[8]. Le BOM défini le système d'encodage à utiliser.

UTF-16LE

De nombreuses fonctions win32 de Windows ont le suffixes `-A` et `-W`. Le premier type de fonctions fonctionne avec les chaînes normales, l'autre, avec des chaîne UTF-16LE (*large*).

Dans le second cas, chaque symbole est en général stocké dans une valeur 16-bit de type *short*.

Les symboles Latin dans les chaînes UTF-16 dans Hiew ou FAR semblent être séparés avec un octet zéro:

```
int wmain()
{
        wprintf (L"Hello, world!\n");
};
```

[8] Byte Order Mark

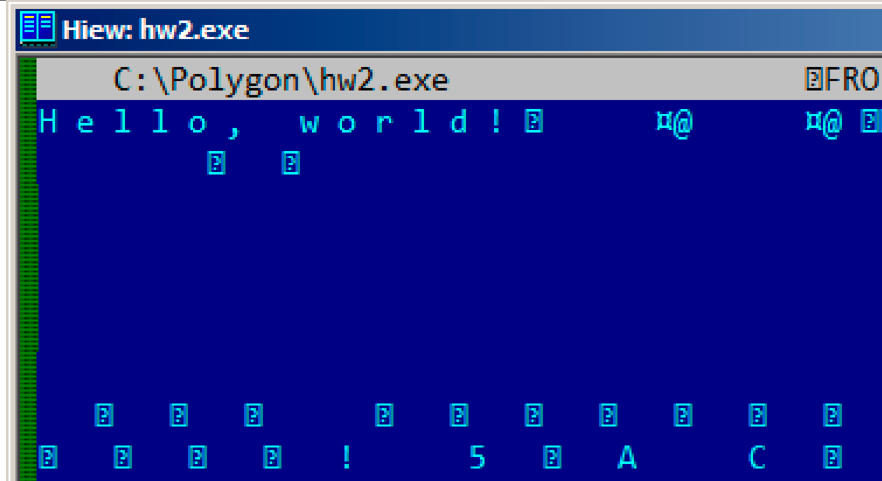


Fig. 5.3: Hiew

Nous voyons souvent ceci dans les fichiers système de Windows NT :

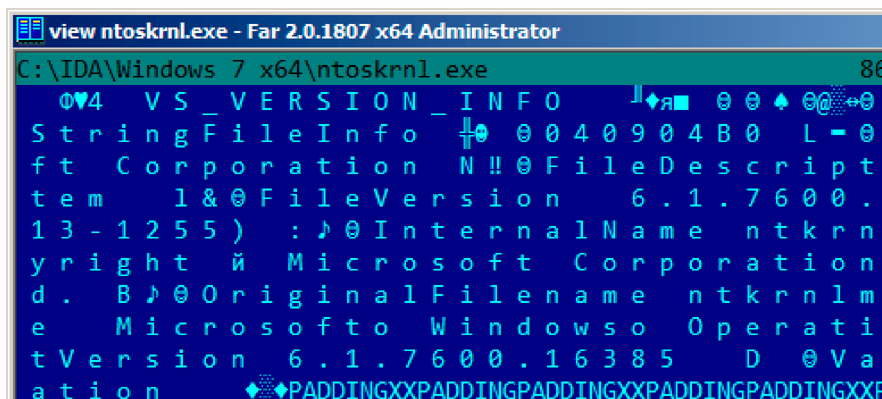


Fig. 5.4: Hiew

Les chaînes avec des caractères qui occupent exactement 2 octets sont appelées «Unicode» dans IDA :

```
.data:0040E000 aHelloWorld:
.data:0040E000                 unicode 0, <Hello, world!>
.data:0040E000                 dw 0Ah, 0
```

Voici comment une chaîne en russe est encodée en UTF-16LE:

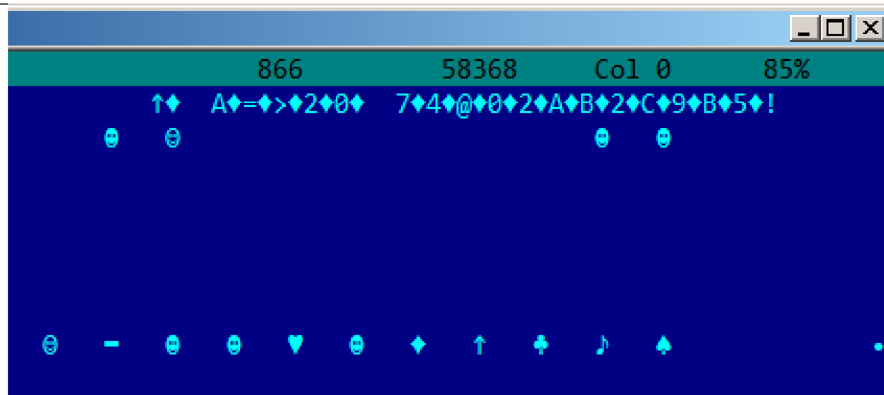


Fig. 5.5: Hiew: UTF-16LE

Ce que nous remarquons facilement, c'est que les symboles sont intercalés par le caractère diamant (qui a le code ASCII 4). En effet, les symboles cyrilliques sont situés dans le quatrième plan Unicode. Ainsi, tous les symboles cyrillique en UTF-16LE sont situés dans l'intervalle `0x400-0x4FF`.

Retournons à l'exemple avec la chaîne écrite dans de multiple langages. Voici à quoi elle ressemble en UTF-16LE.

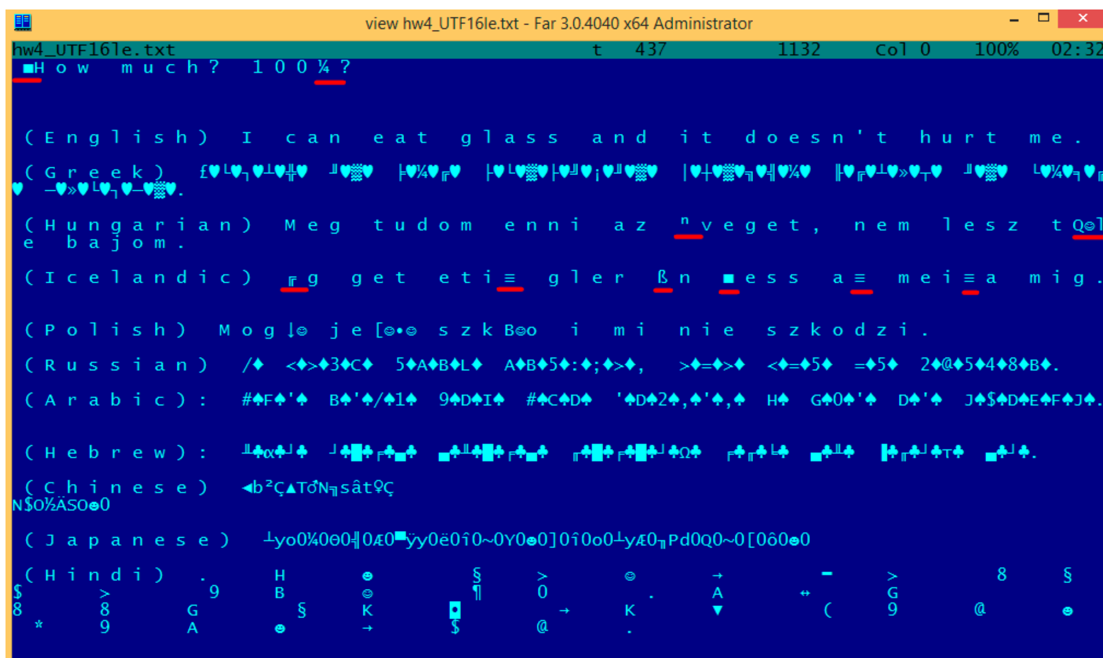


Fig. 5.6: FAR: UTF-16LE

Ici nous pouvons aussi voir le BOM au début. Tous les caractères Latin sont intercalés

avec un octet à zéro.

Certains caractères avec signe diacritique (hongrois et islandais) sont aussi soulignés en rouge.

Base64

L'encodage base64 est très répandu dans les cas où vous devez transférer des données binaires sous forme de chaîne de texte.

Pour l'essentiel, cet algorithme encode 3 octets binaires en 4 caractères imprimables: toutes les 26 lettres Latin (à la fois minuscule et majuscule), chiffres, signe plus («+») et signe slash («/»), 64 caractères en tout.

Une particularité des chaînes base64 est qu'elles se terminent souvent (mais pas toujours) par 1 ou 2 symbole égal («=») pour l'alignement, par exemple:

```
AVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lAOuWs=
```

```
WVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lAOuQ==
```

Le signe égal («=») ne se rencontre jamais au milieu des chaînes encodées en base64.

maintenant, un exemple d'encodage manuel. Encodons les octets hexadécimaux 0x00, 0x11, 0x22, 0x33 en une chaîne base64:

```
$ echo -n "\x00\x11\x22\x33" | base64
ABEiMw==
```

Mettons ces 4 octets au forme binaire, puis regroupons les dans des groupes de 6-bit:

```
|  00  ||  11  ||  22  ||  33  ||      ||      |
00000000000100010010001000110011????????????????
| A  || B  || E  || i  || M  || w  || =  || =  |
```

Les trois premiers octets (0x00, 0x11, 0x22) peuvent être encodés dans 4 caractères base64 ("ABEi"), mais le dernier (0x33) — ne le peut pas, donc il est encodé en utilisant deux caractères ("Mw") et de symbole ("=") de padding est ajouté deux fois pour compléter le dernier groupe à 4 caractères. De ce fait, la longueur de toutes les chaînes en base64 correctes est toujours divisible par 4.

Base64 est souvent utilisé lorsque des données binaires doivent être stockées dans du XML. Les clefs PGP "Armored" (i.e., au format texte) et les signatures sont encodées en utilisant base64.

Certains essayent d'utiliser base64 pour masquer des chaînes: http://blog.sec-consult.com/2016/01/deliberately-hidden-backdoor-account-in.html[9].

[9]http://archive.is/nDCas

Il existe des utilitaires pour rechercher des chaînes base64 dans des fichiers binaires arbitraires. L'un d'entre eux est base64scanner[10].

Un autre système d'encodage qui était très répandu sur UseNet et FidoNet est l'Uuencoding. Les fichiers binaires sont toujours encodés au format Uuencode dans le magazine Phrack. Il offre à peu près la même fonctionnalité, mais il est différent de base64 dans le sens où le nom de fichier est aussi stocké dans l'entête.

À propos: base64 à un petit frère: base32, alphabet qui a 10 chiffres et 26 caractères Latin. Un usage répandu est les adresses onion[11], comme: `http://3g2upl4pq6kufc4m.onion/`. URL ne peut pas avoir de mélange de casse de caractères Latin, donc, c'est apparemment pourquoi les développeurs de Tor ont utilisé base32.

5.4.2 Trouver des chaînes dans un binaire

> Actually, the best form of Unix documentation is frequently running the **strings** command over a program's object code. Using **strings**, you can get a complete list of the program's hard-coded file name, environment variables, undocumented options, obscure error messages, and so forth.
>
> The Unix-Haters Handbook

En fait, la meilleure forme de documentation Unix est de lancer la commande **strings** sur le code objet d'un programme. En utilisant **strings**, vous obtenez une liste complète des noms de fichiers codés en dur dans le programme, les variables d'environnement, les options non documentées, les messages d'erreurs méconnus et ainsi de suite.

L'utilitaire standard UNIX *strings* est un moyen rapide et facile de voir les chaînes dans un fichier. Par exemple, voici quelques chaînes du fichier exécutable sshd d'OpenSSH 7.2:

```
...
0123
0123456789
0123456789abcdefABCDEF.:/
%02x
...
%.100s, line %lu: Bad permitopen specification <%.100s>
%.100s, line %lu: invalid criteria
%.100s, line %lu: invalid tun device
...
%.200s/.ssh/environment
...
2886173b9c9b6fdbdeda7a247cd636db38deaa.debug
$2a$06$r3.juUaHZDlIbQa02dS9FuYxL1W9M81R1Tc92PoSNmzvpEqLkLGrK
```

[10] https://github.com/DennisYurichev/base64scanner
[11] https://trac.torproject.org/projects/tor/wiki/doc/HiddenServiceNames

```
...
3des-cbc
...
Bind to port %s on %s.
Bind to port %s on %s failed: %.200s.
/bin/login
/bin/sh
/bin/sh /etc/ssh/sshrc
...
D$4PQWR1
D$4PUj
D$4PV
D$4PVj
D$4PW
D$4PWj
D$4X
D$4XZj
D$4Y
...
diffie-hellman-group-exchange-sha1
diffie-hellman-group-exchange-sha256
digests
D$iPV
direct-streamlocal
direct-streamlocal@openssh.com
...
FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A6...
...
```

Il y a des options, des messages d'erreur, des chemins de fichier, des modules et des fonctions importés dynamiquement, ainsi que d'autres chaînes étranges (clefs?). Il y a aussi du bruit illisible—le code x86 à parfois des fragments constitués de caractères ASCII imprimables, jusqu'à 8 caractères.

Bien sûr, OpenSSH est un programme open-source. Mais regarder les chaînes lisibles dans un binaire inconnu est souvent une première étape d'analyse.

grep peut aussi être utilisé.

Hiew a la même capacité (Alt-F6), ainsi que ProcessMonitor de Sysinternals.

5.4.3 Messages d'erreur/de débogage

Les messages de débogage sont très utiles s'il sont présents. Dans un certain sens, les messages de débogage rapportent ce qui est en train de se passer dans le programme. Souvent, ce sont des fonctions `printf()`-like, qui écrivent des fichiers de log, ou parfois elles n'écrivent rien du tout mais les appels sont toujours présents puisque le build n'est pas un de débogage mais de *release*.

Si des variables locales ou globales sont affichées dans les messages, ça peut être aussi utile, puisqu'il est possible d'obtenir au moins le nom de la variable. Par exemple, une telle fonction dans Oracle RDBMS est `ksdwrt()`.

Des chaînes de texte significatives sont souvent utiles. Le dés-assembleur IDA peut montrer depuis quelles fonctions et depuis quel endroit cette chaîne particulière est utilisée. Des cas drôles arrivent parfois[12].

Le message d'erreur peut aussi nous aider. Dans Oracle RDBMS, les erreurs sont rapportées en utilisant un groupe de fonctions.
Vous pouvez en lire plus ici: blog.yurichev.com.

Il est possible de trouver rapidement quelle fonction signale une erreur et dans quelles conditions.

À propos, ceci est souvent la raison pour laquelle les systèmes de protection contre la copie utilisent des messages d'erreur inintelligibles ou juste des numéros d'erreur. Personne n'est content lorsque le copieur de logiciel comprend comment fonctionne la protection contre la copie seulement en lisant les messages d'erreur.

Un exemple de messages d'erreur chiffrés se trouve ici: 8.8.2 on page 1091.

5.4.4 Chaînes magiques suspectes

Certaines chaînes magique sont d'habitude utilisées dans les porte dérobées semblent vraiment suspectes.

Par exemple, il y avait une porte dérobée dans le routeur personnel TP-Link WR740[13].
La porte dérobée était activée en utilisant l'URL suivante:
`http://192.168.0.1/userRpmNatDebugRpm26525557/start_art.html`.

En effet, la chaîne «userRpmNatDebugRpm26525557» est présente dans le firmware.

Cette chaîne n'était pas googlable jusqu'à la large révélation d'information concernant la porte dérobée.

Vous ne trouverez ceci dans aucun RFC[14].

Vous ne trouverez pas d'algorithme informatique qui utilise une séquence d'octets aussi étrange.

Et elle ne ressemble pas à une erreur ou un message de débogage.

Donc, c'est une bonne idée d'inspecter l'utilisation de ce genre de chaînes bizarres.

Parfois, de telles chaînes sont encodées en utilisant base64.

Donc, c'est une bonne idée de toutes les décoder et de les inspecter visuellement, même un coup d'œil doit suffire.

Plus précisément, cette méthode de cacher des accès non documentés est appelée «sécurité par l'obscurité».

[12] blog.yurichev.com
[13] `http://sekurak.pl/tp-link-httptftp-backdoor/`
[14] Request for Comments

5.5 Appels à assert()

Parfois, la présence de la macro `assert()` est aussi utile: En général, cette macro laisse le nom du fichier source, le numéro de ligne et une condition dans le code.

L'information la plus utile est contenue dans la condition d'assert, nous pouvons en déduire les noms de variables ou les noms de champ de la structure. Les autres informations utiles sont les noms de fichier—nous pouvons essayer d'en déduire le type de code dont il s'agit ici. Il est aussi possible de reconnaître les bibliothèques open-source connues d'après les noms de fichier.

Listing 5.2 : Exemple d'appels à assert() informatifs

```
.text:107D4B29 mov  dx, [ecx+42h]
.text:107D4B2D cmp  edx, 1
.text:107D4B30 jz   short loc_107D4B4A
.text:107D4B32 push 1ECh
.text:107D4B37 push offset aWrite_c ; "write.c"
.text:107D4B3C push offset aTdTd_planarcon ;
    "td->td_planarconfig == PLANARCONFIG_CON"...
.text:107D4B41 call ds:_assert

...

.text:107D52CA mov  edx, [ebp-4]
.text:107D52CD and  edx, 3
.text:107D52D0 test edx, edx
.text:107D52D2 jz   short loc_107D52E9
.text:107D52D4 push 58h
.text:107D52D6 push offset aDumpmode_c ; "dumpmode.c"
.text:107D52DB push offset aN30     ; "(n & 3) == 0"
.text:107D52E0 call ds:_assert

...

.text:107D6759 mov  cx, [eax+6]
.text:107D675D cmp  ecx, 0Ch
.text:107D6760 jle  short loc_107D677A
.text:107D6762 push 2D8h
.text:107D6767 push offset aLzw_c   ; "lzw.c"
.text:107D676C push offset aSpLzw_nbitsBit ; "sp->lzw_nbits <= BITS_MAX"
.text:107D6771 call ds:_assert
```

Il est recommandé de «googler» à la fois les conditions et les noms de fichier, qui peuvent nous conduire à une bibliothèque open-source. Par exemple, si nous «googlons» «sp->lzw_nbits <= BITS_MAX», cela va comme prévu nous donner du code open-source relatif à la compression LZW.

5.6 Constantes

Les humains, programmeurs inclus, utilisent souvent des nombres ronds, comme 10, 100, 1000, dans la vie courante comme dans le code.

Le rétro ingénieur pratiquant connaît en général bien leur représentation décimale: 10=0xA, 100=0x64, 1000=0x3E8, 10000=0x2710.

Les constantes `0xAAAAAAAA` (0b10101010101010101010101010101010) et `0x55555555` (0b01010101010101010101010101010101) sont aussi répandues—elles sont composées d'alternance de bits.

Cela peut aider à distinguer un signal d'un signal dans lequel tous les bits sont à 1 (0b1111 …) ou à 0 (0b0000 …). Par exemple, la constante `0x55AA` est utilisée au moins dans le secteur de boot, MBR[15], et dans la ROM de cartes d'extention de compatible IBM.

Certains algorithmes, particulièrement ceux de chiffrement, utilisent des constantes distinctes, qui sont faciles à trouver dans le code en utilisant IDA.

Par exemple, l'algorithme MD5 initialise ses propres variables internes comme ceci:

```
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

Si vous trouvez ces quatre constantes utilisées à la suite dans du code, il est très probable que cette fonction soit relatives à MD5.

Un autre exemple sont les algorithmes CRC16/CRC32, ces algorithmes de calcul utilisent souvent des tables pré-calculées comme celle-ci:

Listing 5.3 : linux/lib/crc16.c

```
/** CRC table for the CRC-16. The poly is 0x8005 (x^16 + x^15 + x^2 + 1) */
u16 const crc16_table[256] = {
        0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
        0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
        0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
        ...
```

Voir aussi la table pré-calculée pour CRC32: 3.8 on page 615.

Dans les algorithmes CRC sans table, des polynômes bien connus sont utilisés, par exemple 0xEDB88320 pour CRC32.

5.6.1 Nombres magiques

De nombreux formats de fichier définissent un entête standard où un *nombre(s) magique* est utilisé, unique ou même plusieurs.

Par exemple, tous les exécutables Win32 et MS-DOS débutent par ces deux caractères «MZ»[16].

Au début d'un fichier MIDI, la signature «MThd» doit être présente. Si nous avons un programme qui utilise des fichiers MIDI pour quelque chose, il est très probable qu'il doit vérifier la validité du fichier en testant au moins les 4 premiers octets.

[15] Master Boot Record
[16] Wikipédia

Ça peut être fait comme ceci: (*buf* pointe sur le début du fichier chargé en mémoire)

```
cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file
```

…ou en appelant une fonction pour comparer des blocs de mémoire comme memcmp() ou tout autre code équivalent jusqu'à une instruction CMPSB (.1.6 on page 1338).

Lorsque vous trouvez un tel point, vous pouvez déjà dire que le chargement du fichier MIDI commence, ainsi, vous pouvez voir l'endroit où se trouve le buffer avec le contenu du fichier MIDI, ce qui est utilisé dans le buffer et comment.

Dates

Souvent, on peut rencontrer des nombres comme 0x19870116, qui ressemble clairement à une date (année 1987, 1er mois (janvier), 16ème jour). Ça peut être la date de naissance de quelqu'un (un programmeur, une de ses relations, un enfant), ou une autre date importante. La date peut aussi être écrite dans l'ordre inverse, comme 0x16011987. Les dates au format américain sont aussi courante, comme 0x01161987.

Un exemple célèbre est 0x19540119 (nombre magique utilisé dans la structure du super-bloc UFS2), qui est la date de naissance de Marshall Kirk McKusick, éminent contributeur FreeBSD.

Stuxnet utilise le nombre "19790509" (pas comme un nombre 32-bit, mais comme une chaîne, toutefois), et ça a conduit à spéculer que le malware était relié à Israël[17].

Aussi, des nombres comme ceux-ci sont très répandus dans dans le chiffrement niveau amateur, par exemple, extrait de la *fonction secrète* des entrailles du dongle HASP3[18] :

```
void xor_pwd(void)
{
        int i;

        pwd^=0x09071966;
        for(i=0;i<8;i++)
        {
                al_buf[i]= pwd & 7; pwd = pwd >> 3;
        }
};

void emulate_func2(unsigned short seed)
{
        int i, j;
        for(i=0;i<8;i++)
        {
                ch[i] = 0;

                for(j=0;j<8;j++)
```

[17] C'est la date d'exécution de Habib Elghanian, juif persan.
[18] https://web.archive.org/web/20160311231616/http://www.woodmann.com/fravia/bayu3.htm

```
                {
                        seed *= 0x1989;
                        seed += 5;
                        ch[i] |= (tab[(seed>>9)&0x3f]) << (7-j);
                }
        }
}
```

DHCP

Ceci s'applique aussi aux protocoles réseaux. Par exemple, les paquets réseau du protocole DHCP contiennent un soi-disant *nombre magique* : 0x63538263. Tout code qui génère des paquets DHCP doit contenir quelque part cette constante à insérer dans les paquets. Si nous la trouvons dans du code, nous pouvons trouver ce qui s'y passe, et pas seulement ça. Tout programme qui peut recevoir des paquet DHCP doit vérifier le *cookie magique*, et le comparer à cette constante.

Par exemple, prenons le fichier dhcpcore.dll de Windows 7 x64 et cherchons cette constante. Et nous la trouvons, deux fois: Il semble que la constante soit utilisée dans deux fonctions avec des noms parlants
DhcpExtractOptionsForValidation() et DhcpExtractFullOptions() :

Listing 5.4 : dhcpcore.dll (Windows 7 x64)

```
.rdata:000007FF6483CBE8 dword_7FF6483CBE8 dd 63538263h          ; DATA XREF:
    DhcpExtractOptionsForValidation+79
.rdata:000007FF6483CBEC dword_7FF6483CBEC dd 63538263h          ; DATA XREF:
    DhcpExtractFullOptions+97
```

Et ici sont les endroits où ces constantes sont accédées:

Listing 5.5 : dhcpcore.dll (Windows 7 x64)

```
.text:000007FF6480875F  mov     eax, [rsi]
.text:000007FF64808761  cmp     eax, cs:dword_7FF6483CBE8
.text:000007FF64808767  jnz     loc_7FF64817179
```

Et:

Listing 5.6 : dhcpcore.dll (Windows 7 x64)

```
.text:000007FF648082C7  mov     eax, [r12]
.text:000007FF648082CB  cmp     eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1  jnz     loc_7FF648173AF
```

5.6.2 Constantes spécifiques

Parfois, il y a une constante spécifique pour un certain type de code. Par exemple, je me suis plongé une fois dans du code, où le nombre 12 était rencontré anormalement souvent. La taille de nombreux tableaux était 12 ou un multiple de 12 (24, etc.). Il s'est avéré que ce code prenait des fichiers audio de 12 canaux en entrée et les traitait.

Et vice versa: par exemple, si un programme fonctionne avec des champs de texte qui ont une longueur de 120 octets, il doit y avoir une constante 120 ou 119 quelque part dans le code. Si UTF-16 est utilisé, alors $2 \cdot 120$. Si le code fonctionne avec des paquets réseau de taille fixe, c'est une bonne idée de chercher cette constante dans le code.

C'est aussi vrai pour le chiffrement amateur (clefs de licence, etc.). Si le bloc chiffré a une taille de n octets, vous pouvez essayer de trouver des occurrences de ce nombre à travers le code. Aussi, si vous voyez un morceau de code qui est répété n fois dans une boucle durant l'exécution, ceci peut être une routine de chiffrement/déchiffrement.

5.6.3 Chercher des constantes

C'est facile dans IDA : Alt-B or Alt-I. Et pour chercher une constante dans un grand nombre de fichiers, ou pour chercher dans des fichiers non exécutables, il y a un petit utilitaire appelé *binary grep*[19].

5.7 Trouver les bonnes instructions

Si le programme utilise des instructions FPU et qu'il n'y en a que quelques une dans le code, on peut essayer de les vérifier chacunes manuellement avec un débogueur.

Par exemple, nous pouvons être intéressés de comprendre comment Microsoft Excel calcule la formule entrée par l'utilisateur. Par exemple, l'opération de division.

Si nous chargeons excel.exe (d'Office 2010) version 14.0.4756.1000 dans IDA, faisons un listing complet et cherchons chaque instruction FDIV (sauf celle qui utilisent une constante comme second opérande—évidemment, elles ne nous intéressent pas) :

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

…nous voyons alors qu'il y en a 144.

Nous pouvons entrer une chaîne comme =(1/3) dans Excel et vérifier chaque instruction.

En vérifiant chaque instruction dans un débogueur ou tracer (on peut vérifier 4 instructions à la fois), nous avons de la chance et l'instruction que nous cherchons n'est que la 14ème:

```
.text:3011E919 DC 33          fdiv    qword ptr [ebx]
```

```
PID=13944|TID=28744|(0) 0x2f64e919 (Excel.exe!BASE+0x11e919)
EAX=0x02088006 EBX=0x02088018 ECX=0x00000001 EDX=0x00000001
ESI=0x02088000 EDI=0x00544804 EBP=0x0274FA3C ESP=0x0274F9F8
EIP=0x2F64E919
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
```

[19] GitHub

```
FPU StatusWord=
FPU ST(0): 1.000000
```

ST(0) contient le premier argument (1) et le second est dans [EBX].

L'instruction après FDIV (FSTP) écrit le résultat en mémoire:

```
.text:3011E91B DD 1E          fstp    qword ptr [esi]
```

Si nous mettons un point d'arrêt dessus, nous voyons le résultat:

```
PID=32852|TID=36488|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00598006 EBX=0x00598018 ECX=0x00000001 EDX=0x00000001
ESI=0x00598000 EDI=0x00294804 EBP=0x026CF93C ESP=0x026CF8F8
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
```

Pour blaguer, nous pouvons modifier le résultat au vol:

```
tracer -l:excel.exe bpx=excel.exe!BASE+0x11E91B,set(st0,666)
```

```
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000 EDI=0x00395404 EBP=0x0290FD9C ESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

Excel affiche 666 dans la cellule, achevant de nous convaincre que nous avons trouvé le bon endroit.

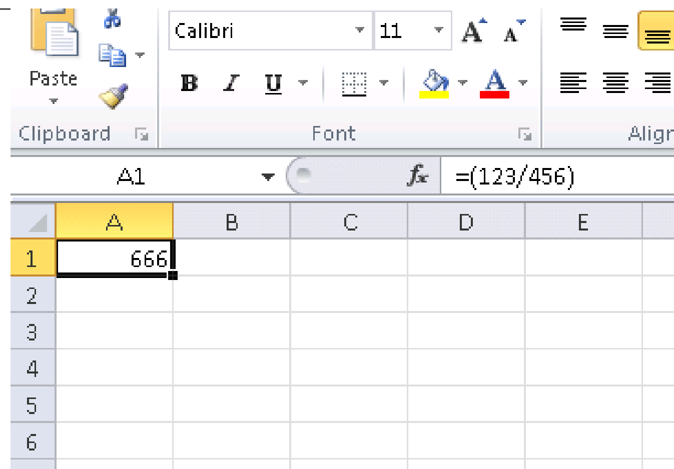


Fig. 5.7: La blague a fonctionné

Si nous essayons la même version d'Excel, mais en x64, nous allons y trouver seulement 12 instructions FDIV, et celle que nous cherchons est la troisième.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC,set(st0,666)
```

Il semble que le compilateur a remplacé beaucoup d'opérations de division de types *float* et *double*, par des instructions SSE comme DIVSD (DIVSD est présent 268 fois en tout).

5.8 Patterns de code suspect

5.8.1 instructions XOR

Des instructions comme XOR op, op (par exemple, XOR EAX, EAX) sont utilisées en général pour mettre la valeur d'un registre à zéro, mais si les opérandes sont différentes, l'opération «ou exclusif» est exécutée.

Cette opération est rare en programmation courante, mais répandu en cryptographie, y compris amateur. C'est particulièrement suspect si le second opérande est un grand nombre.

Ceci peut indiquer du chiffrement/déchiffrement, du calcul de somme de contrôle, etc.

Une exception à cette observation, qu'il est utile de noter, est le «canari» (1.26.3 on page 358). Sa génération et sa vérification sont souvent effectuées en utilisant des instructions XOR.

Ce script awk peut être utilisé pour traité les fichiers listing (.lst) d'IDA :

```
gawk -e '$2=="xor" { tmp=substr($3, 0, length($3)-1); if (tmp!=$4) if($4!="↙
    ↘ esp") if ($4!="ebp") { print $1, $2, tmp, ",", $4 } }' filename.lst
```

Il est aussi utile de noter que ce type de script peut aussi rapporter du code mal désassemblé (5.11.1 on page 944).

5.8.2 Code assembleur écrit à la main

Les compilateurs modernes ne génèrent pas les instructions LOOP et RCL. D'un autre côté, ces instructions sont très connues des codeurs qui aiment écrire directement en langage d'assemblage. Si vous les rencontrez, on peut dire qu'il est très probable que ce morceau de code ait été écrit à la main. De telles instructions sont marquées avec un (M) dans la liste des instructions de cet appendice: .1.6 on page 1330.

De même, les prologue/épilogue de fonction sont rares dans de l'assembleur écrit à la main.

Il n'y a généralement pas de système fixé pour le passage des arguments aux fonctions dans du code écrit à la main.

Exemple du noyau de Windows 2003 (ntoskrnl.exe file) :

```
MultiplyTest proc near               ; CODE XREF: Get386Stepping
             xor     cx, cx
loc_620555:                          ; CODE XREF: MultiplyTest+E
             push    cx
             call    Multiply
             pop     cx
             jb      short locret_620563
             loop    loc_620555
             clc
locret_620563:                       ; CODE XREF: MultiplyTest+C
             retn
MultiplyTest endp

Multiply     proc near               ; CODE XREF: MultiplyTest+5
             mov     ecx, 81h
             mov     eax, 417A000h
             mul     ecx
             cmp     edx, 2
             stc
             jnz     short locret_62057F
             cmp     eax, 0FE7A000h
             stc
             jnz     short locret_62057F
             clc
locret_62057F:                       ; CODE XREF: Multiply+10
                                     ; Multiply+18
             retn
Multiply     endp
```

En effet, si nous regardons dans le code source de WRK[20] v1.2, ce code peut être trouvé facilement dans le fichier
WRK-v1.2\base\ntos\ke\i386\cpu.asm.

D'après l'instruction RCL que j'ai pu trouver dans le fichier ntoskrnl.exe de Windows 2003 x86 (compilé avec MS Visual C compiler). Elle apparaît seulement une fois ici, dans la fonction `RtlExtendedLargeIntegerDivide()` et ça pourrait être un cas de code assembleur en ligne.

5.9 Utilisation de nombres magiques lors du tracing

Souvent, notre but principal est de comprendre comment le programme utilise une valeur qui a été soit lue d'un fichier ou reçue par le réseau. Le tracing manuel d'une valeur est souvent une tâche laborieuse. Une des techniques les plus simple pour ceci (bien que non sûre à 100%) est d'utiliser votre propre *nombre magique*.

Ceci ressemble à la tomodensitométrie aux rayons X: un agent de radio-contraste est injecté dans le sang du patient, qui est utilisé pour augmenter la visibilité de la structure interne du patient aux rayons X. C'est bien connu comment le sang circule dans les reins d'humains en bonne santé et si l'agent est dans le sang, il peut être vu facilement en tomographie comment le sang circule et si il y a des calculs ou des tumeurs.

Nous pouvons prendre un nombre 32-bit comme `0x0badf00d`, ou la date de naissance de quelqu'un comme `0x11101979` et écrire ce nombre de 4 octets quelque part dans un fichier utilisé par le programme que nous investiguons.

Puis, en suivant ce programme avec tracer en mode *code coverage*, avec l'aide de *grep* ou simplement en cherchant dans le fichier texte (résultant de l'investigation), nous pouvons facilement voir où la valeur a été utilisée et comment.

Exemple de résultats de tracer *grepable* en mode *cc* :

```
0x150bf66 (_kziaia+0x14), e=       1 [MOV EBX, [EBP+8]] [EBP+8]=0xf59c934
0x150bf69 (_kziaia+0x17), e=       1 [MOV EDX, [69AEB08h]] [69AEB08h]=0
0x150bf6f (_kziaia+0x1d), e=       1 [FS: MOV EAX, [2Ch]]
0x150bf75 (_kziaia+0x23), e=       1 [MOV ECX, [EAX+EDX*4]] [EAX+EDX*4]=0↙
    ↳ xf1ac360
0x150bf78 (_kziaia+0x26), e=       1 [MOV [EBP-4], ECX] ECX=0xf1ac360
```

Cela peut aussi être utilisé pour des paquets réseau. Il est important que le *nombre magique* soit unique et ne soit pas présent dans le code du programme.

À part tracer, DosBox (émulateur MS-DOS) en mode heavydebug est capable d'écrire de l'information à propos de l'état de tous les registres pour chaque instruction du programme exécutée dans un fichier texte[21], donc cette technique peut être utile également pour des programmes DOS.

[20]Windows Research Kernel
[21]Voir aussi mon article de blog sur cette fonctionnalité de DosBox: blog.yurichev.com

5.10 Boucles

À chaque fois que votre programme travaille avec des sortes de fichier, ou un buffer d'une certaine taille, il doit s'agir d'un sorte de boucle de déchiffrement/traitement à l'intérieur du code.

Ceci est un exemple réel de sortie de l'outil tracer. Il y avait un code qui chargeait une sorte de fichier chiffré de 258 octets. Je l'ai lancé dans l'intention d'obtenir le nombre d'exécution de chaque instruction (l'outil DBI irait beaucoup mieux de nos jours). Et j'ai rapidement trouvé un morceau de code qui était exécuté 259/258 fois:

```
...

0x45a6b5 e=   1 [FS: MOV [0], EAX] EAX=0x218fb08
0x45a6bb e=   1 [MOV [EBP-254h], ECX] ECX=0x218fbd8
0x45a6c1 e=   1 [MOV EAX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a6c7 e=   1 [CMP [EAX+14h], 0] [EAX+14h]=0x102
0x45a6cb e=   1 [JZ 45A9F2h] ZF=false
0x45a6d1 e=   1 [MOV [EBP-0Dh], 1]
0x45a6d5 e=   1 [XOR ECX, ECX] ECX=0x218fbd8
0x45a6d7 e=   1 [MOV [EBP-14h], CX] CX=0
0x45a6db e=   1 [MOV [EBP-18h], 0]
0x45a6e2 e=   1 [JMP 45A6EDh]
0x45a6e4 e= 258 [MOV EDX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0↙
    ↳ xfd..0x101
0x45a6e7 e= 258 [ADD EDX, 1] EDX=0..5 (248 items skipped) 0xfd..0x101
0x45a6ea e= 258 [MOV [EBP-18h], EDX] EDX=1..6 (248 items skipped) 0xfe..0↙
    ↳ x102
0x45a6ed e= 259 [MOV EAX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a6f3 e= 259 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (249 items skipped) 0↙
    ↳ xfe..0x102
0x45a6f6 e= 259 [CMP ECX, [EAX+14h]] ECX=0..5 (249 items skipped) 0xfe..0↙
    ↳ x102 [EAX+14h]=0x102
0x45a6f9 e= 259 [JNB 45A727h] CF=false,true
0x45a6fb e= 258 [MOV EDX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a701 e= 258 [MOV EAX, [EDX+10h]] [EDX+10h]=0x21ee4c8
0x45a704 e= 258 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0↙
    ↳ xfd..0x101
0x45a707 e= 258 [ADD ECX, 1] ECX=0..5 (248 items skipped) 0xfd..0x101
0x45a70a e= 258 [IMUL ECX, ECX, 1Fh] ECX=1..6 (248 items skipped) 0xfe..0↙
    ↳ x102
0x45a70d e= 258 [MOV EDX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0↙
    ↳ xfd..0x101
0x45a710 e= 258 [MOVZX EAX, [EAX+EDX]] [EAX+EDX]=1..6 (156 items skipped) 0↙
    ↳ xf3, 0xf8, 0xf9, 0xfc, 0xfd
0x45a714 e= 258 [XOR EAX, ECX] EAX=1..6 (156 items skipped) 0xf3, 0xf8, 0↙
    ↳ xf9, 0xfc, 0xfd ECX=0x1f, 0x3e, 0x5d, 0x7c, 0x9b (248 items skipped) ↙
    ↳ 0x1ec2, 0x1ee1, 0x1f00, 0x1f1f, 0x1f3e
0x45a716 e= 258 [MOV ECX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a71c e= 258 [MOV EDX, [ECX+10h]] [ECX+10h]=0x21ee4c8
0x45a71f e= 258 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0↙
    ↳ xfd..0x101
0x45a722 e= 258 [MOV [EDX+ECX], AL] AL=0..5 (77 items skipped) 0xe2, 0xee, ↙
```

```
    ↳ 0xef, 0xf7, 0xfc
0x45a725 e= 258 [JMP 45A6E4h]
0x45a727 e=   1 [PUSH 5]
0x45a729 e=   1 [MOV ECX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a72f e=   1 [CALL 45B500h]
0x45a734 e=   1 [MOV ECX, EAX] EAX=0x218fbd8
0x45a736 e=   1 [CALL 45B710h]
0x45a73b e=   1 [CMP EAX, 5] EAX=5

...
```

Il s'avère qu'il s'agit de la boucle de déchiffrement.

5.10.1 Quelques schémas de fichier binaire

Tous les exemples ici ont été préparé sur Windows, avec la page de code 437 activéedans la console. L'intérieur des fichiers binaires peut avoir l'air différent avec une autre page de code.

Tableaux

Parfois, nous pouvons clairement localiser visuellement un tableau de valeurs 16/32/64-bit, dans un éditeur hexadécimal.

Voici un exemple de tableau de valeurs 16-bit. Nous voyons que le premier octet d'une paire est 7 ou 8, et que le second semble aléatoire:

```
E:\...3affacde09fe21c28f1543db51145b.dat h  1252        2175000     Col 0      23%   21:25
000007CA70: EF 07 C6 07 D6 07 26 08  0C 08 CE 07 24 07 60 07
000007CA80: CC 07 AA 07 A2 07 AC 07  E9 07 BF 07 D6 07 2C 08
000007CA90: 09 08 CA 07 31 07 5E 07  BC 07 9A 07 93 07 9E 07
000007CAA0: E6 07 BD 07 D8 07 2F 08  06 08 CB 07 3E 07 5E 07
000007CAB0: B3 07 91 07 8B 07 97 07  E1 07 BB 07 DB 07 32 08
000007CAC0: 03 08 CB 07 4C 07 61 07  AA 07 89 07 84 07 91 07
000007CAD0: E0 07 BB 07 DC 07 33 08  01 08 CC 07 57 07 64 07
000007CAE0: A4 07 84 07 81 07 90 07  DE 07 BB 07 DE 07 34 08
000007CAF0: FF 07 CD 07 65 07 69 07  A0 07 81 07 7F 07 90 07
000007CB00: DE 07 BC 07 DF 07 33 08  FF 07 CE 07 70 07 6F 07
000007CB10: 9F 07 82 07 81 07 93 07  DD 07 BC 07 E0 07 34 08
000007CB20: FE 07 CE 07 7E 07 78 07  9F 07 84 07 84 07 96 07
000007CB30: DE 07 BD 07 DF 07 32 08  FF 07 CE 07 87 07 7F 07
000007CB40: A1 07 87 07 88 07 9B 07  E2 07 BF 07 DE 07 2F 08
000007CB50: 02 08 CF 07 93 07 89 07  A4 07 8C 07 8D 07 9F 07
000007CB60: E4 07 C0 07 DD 07 2D 08  03 08 CF 07 9C 07 92 07
000007CB70: A9 07 90 07 91 07 A3 07  E6 07 C3 07 DD 07 2B 08
000007CB80: 04 08 D0 07 A7 07 9C 07  AE 07 96 07 96 07 A7 07
000007CB90: E8 07 C7 07 DF 07 29 08  04 08 D3 07 B1 07 A7 07
000007CBA0: B4 07 9B 07 9B 07 AB 07  E8 07 CA 07 E1 07 27 08
000007CBB0: 03 08 D5 07 BB 07 B3 07  BB 07 A1 07 A0 07 AF 07
000007CBC0: EA 07 CD 07 E3 07 25 08  03 08 D8 07 C4 07 BD 07
000007CBD0: C1 07 A6 07 A5 07 B3 07  EA 07 D1 07 E6 07 22 08
000007CBE0: 01 08 DC 07 CE 07 C8 07  C8 07 AD 07 AA 07 B7 07
1Help  2Wrap  3Quit  4Text  5  6Edit  7Search 8OEM  9  10Quit
```

Fig. 5.8: FAR: tableau de valeurs 16-bit

J'ai utilisé un fichier contenant un signal 12-canaux numérisé en utilisant 16-bit ADC[22].

[22] Analog-to-Digital Converter

Et voici un exemple de code MIPS très typique.

Comme nous pouvons nous en souvenir, chaque instruction MIPS (et aussi ARM en mode ARM ou ARM64) a une taille de 32 bits (ou 4 octets), donc un tel code est un tableau de valeurs 32-bit.

En regardant cette copie d'écran, nous voyons des sortes de schémas.

Les lignes rouge verticales ont été ajoutées pour la clarté:

```
Hiew: FW96650A.bin
   FW96650A.bin                              FRO --------                 00005000
00005000:  A0 B0 02 3C-04 00 BE AF-40 00 43 8C-21 F0 A0 03
00005010:  FF 1F 02 3C-21 E8 C0 03-FF FF 42 34-24 10 62 00
00005020:  00 A0 03 3C-25 10 43 00-04 00 BE 8F-08 00 E0 03
00005030:  08 00 BD 27-F8 FF BD 27-A0 B0 02 3C-04 00 BE AF
00005040:  48 00 43 8C-21 F0 A0 03-FF 1F 02 3C-21 E8 C0 03
00005050:  FF FF 42 34-24 10 62 00-00 A0 03 3C-25 10 43 00
00005060:  04 00 BE 8F-08 00 E0 03-08 00 BD 27-F8 FF BD 27
00005070:  21 10 00 00-04 00 BE AF-08 00 80 14-21 F0 A0 03
00005080:  A0 B0 03 3C-21 E8 C0 03-44 29 02 7C-3C 00 62 AC
00005090:  04 00 BE 8F-08 00 E0 03-08 00 BD 27-01 00 03 24
000050A0:  44 29 62 7C-A0 B0 03 3C-21 E8 C0 03-3C 00 62 AC
000050B0:  04 00 BE 8F-08 00 E0 03-08 00 BD 27-F8 FF BD 27
000050C0:  A0 B0 02 3C-04 00 BE AF-84 00 43 8C-21 F0 A0 03
000050D0:  21 E8 C0 03-C4 FF 03 7C-84 00 43 AC-04 00 BE 8F
000050E0:  08 00 E0 03-08 00 BD 27-F8 FF BD 27-A0 B0 02 3C
000050F0:  04 00 BE AF-20 00 43 8C-21 F0 A0 03-01 00 04 24
00005100:  21 E8 C0 03-44 08 83 7C-20 00 43 AC-04 00 BE 8F
00005110:  08 00 E0 03-08 00 BD 27-F8 FF BD 27-A0 B0 02 3C
00005120:  04 00 BE AF-20 00 43 8C-21 F0 A0 03-21 E8 C0 03
00005130:  44 08 03 7C-20 00 43 AC-04 00 BE 8F-08 00 E0 03
00005140:  08 00 BD 27-F8 FF BD 27-A0 B0 03 3C-04 00 BE AF
00005150:  10 00 62 8C-01 00 08 24-04 A5 02 7D-08 00 09 24
00005160:  10 00 62 AC-04 7B 22 7D-04 48 02 7C-04 84 02 7D
00005170:  10 00 62 AC-21 F0 A0 03-21 18 00 00-A0 B0 0B 3C
00005180:  51 00 0A 24-02 00 88 94-00 00 89 94-00 44 08 00
00005190:  25 40 09 01-01 00 63 24-14 00 68 AD-F9 FF 6A 14
000051A0:  04 00 84 24-21 18 00 00-A0 B0 0A 3C-07 00 09 24
000051B0:  02 00 A4 94-00 00 A8 94-00 24 04 00-25 20 88 00
1Global 2FilBlk 3CryBlk 4ReLoad 5       6String 7Direct 8Table  9       10Leave 11
```

Fig. 5.9: Hiew: code MIPS très typique

Il y a un autre exemple de tel schéma ici dans le livre: 9.5 on page 1255.

Fichiers clairsemés

Ceci est un fichier clairsemé avec des données éparpillées dans un fichier presque vide. Chaque caractère espace est en fait l'octet zéro (qui rend comme un espace). Ceci est un fichier pour programmer des FPGA (Altera Stratix GX device). Bien sûr, de tels fichiers peuvent être compressés facilement, mais des formats comme celui-ci sont très populaire dans les logiciels scientifiques et d'ingénierie, où l'efficience des accès est importante, tandis que la compacité ne l'est pas.

Z:\...gns\hello_gx_restored\hello_gx.sof d 437 2947940 Col 0 83% 22:11

1Help 2Unwrap 3Quit 4Hex 5 6Edit 7Search 8ANSI 9 10Quit

Fig. 5.10: FAR: Fichier clairsemé

Fichiers compressés

Ce fichier est juste une archive compressée. Il a une entropie relativement haute et visuellement, il à l'air chaotique. Ceci est ce à quoi ressemble les fichiers compressés et/ou chiffrés.

Fig. 5.11: FAR: Fichier compressé

CDFS[23]

Les fichiers d'installation d'un OS sont en général distribués sous forme de fichiers ISO, qui sont des copies de disques CD/DVD. Le système de fichiers utilisé est appelé CDFS, ce que vous voyez ici sont des noms de fichiers mixés avec des données additionnelles. Ceci peut-être la taille des fichiers, des pointeurs sur d'autres répertoires, des attributs de fichier, etc. C'est l'aspect typique de ce à quoi ressemble un système de fichiers en interne.

```
Z:\...untu\ubuntu-15.10-desktop-i386.iso d    437     1226964992      Col 0       0%    21:59

                                                 ä #       # ◘     ◘ s◙§►↕; ☻  ☺  ☺☺ SP•☺╛n
 PX$☺mA     Am☺       ☺                TF→☺♫s◙§►↕, s◙§►↕, s◙§►↕, CE└☺$       $        ф
  ф ` #       # ◘     ◘ s◙§►↕; ☻  ☺  ☺☺☺PX$☺mA     Am☺       ☺                TF→☺♫s◙§►↕, s◙
§►↕, s◙§►↕, n %       % ◘     ◘ s◙§►↕; ☻  ☺  ☺♣.DISKPX$☺mA     Am☺       ☺                TF
→☺♫s◙§►↕% s◙§►↕, s◙§►↕% NM◙☺ .diskn &       & ◘     ◘ s◙§►↕; ☻  ☺  ☺♦BOOT PX$☺mA     Am☺
   ☺                TF→☺♫s◙§►↕( s◙§►↕, s◙§►↕( NM○☺ boot r (       ( ◘     ◘ s◙§►↕; ☻  ☺  ☺
♠CASPER PX$☺mA     Am☺       ☺                TF→☺♫s◙§►↕( s◙§►↕, s◙§►↕( NM♂☺ casper n )
   ) ◘     ◘ s◙§►↕; ☻  ☺  ☺♣DISTSPX$☺mA     Am☺       ☺                TF→☺♫s◙§►↕→ s◙§►↕→ s
◙§►↕→ NM◙☺ distsr 1       1 ◘     ◘ s◙§►↕; ☻  ☺  ☺•INSTALLPX$☺mA     Am☺       ☺
    TF→☺♫s◙§►↕( s◙§►↕, s◙§►↕( NM♀☺ installv 2       2 H     H s◙§►↕; ☻  ☺  ☺◘ISOLINUX PX$☺
mA     Am☺       ☺                TF→☺♫s◙§►↕( s◙§►↕( s◙§►↕( NM♪☺ isolinux z p☻     ☻p`♫
♫`s◙§►↕;     ☺  ☺◙MD5SUM.TXT PX$☺$ü     ü$☺       ☺                TF→☺♫s◙§►↕: s◙§►↕; s◙§►↕
: NM☼☺ md5sum.txt n ;       ; ◘     ◘ s◙§►↕; ☻  ☺  ☺♦PICS PX$☺mA     Am☺       ☺
    TF→☺♫s◙§►↕→ s◙§►↕→ s◙§►↕→ NM○☺ pics n <       < ◘     ◘ s◙§►↕; ☻  ☺  ☺♦POOL PX$☺mA
Am☺       ☺                TF→☺♫s◙§►↕→ s◙§►↕  s◙§►↕→ NM○☺ pool r W       W ◘     ◘ s◙§►↕; ☻
  ☺  ☺•PRESEEDPX$☺mA     Am☺       ☺                TF→☺♫s◙§►↕→ s◙§►↕→ s◙§►↕→ NM♀☺ preseed
è S☻     ☻Sα       αs◙§►↕;     ☺  ☺↕README.DISKDEFINES PX$☺$ü     ü$☺       ☺
TF→☺♫s◙§►↕→ s◙§►↕: s◙§►↕→ NM‡☺ README.diskdefines x ≤       ≤        s◙§►↕;     ☺  ☺•UBUNT
U.PX$☺mí     ím☺       ☺                TF→☺♫s◙§►↕→ s◙§►↕→ s◙§►↕→ NM♂☺ ubuntuSL•☺ ☻



1Help   2Unwrap 3Quit   4Hex    5       6Edit   7Search 8ANSI   9       10Quit
```

Fig. 5.12: FAR: Fichier ISO: CD[24] d'installation d'Ubuntu 15

[23]Compact Disc File System

Code exécutable x86 32-bit

Voici l'allure de code exécutable x86 32-bit. Il n'a pas une grande entropie, car certains octets reviennent plus souvent que d'autres.

Z:\...niversal-USB-Installer-1.9.6.0.exe d 437 1089027 Col 0 0% 22:05

1Help 2Unwrap 3Quit 4Hex 5 6Edit 7Search 8ANSI 9 10Quit

Fig. 5.13: FAR: Code exécutable x86 32-bit

Fichiers graphique BMP

Les fichiers BMP ne sont pas compressés, donc chaque octet (ou groupe d'octet) représente chaque pixel. J'ai trouvé cette image quelque part dans mon installation de Windows 8.1:

Fig. 5.14: Image exemple

Vous voyez que cette image a des pixels qui ne doivent pas pouvoir être compressés beaucoup (autour du centre), mais il y a de longues lignes monochromes au haut et en bas. En effet, de telles lignes ressemblent à des lignes lorsque l'on regarde le fichier:

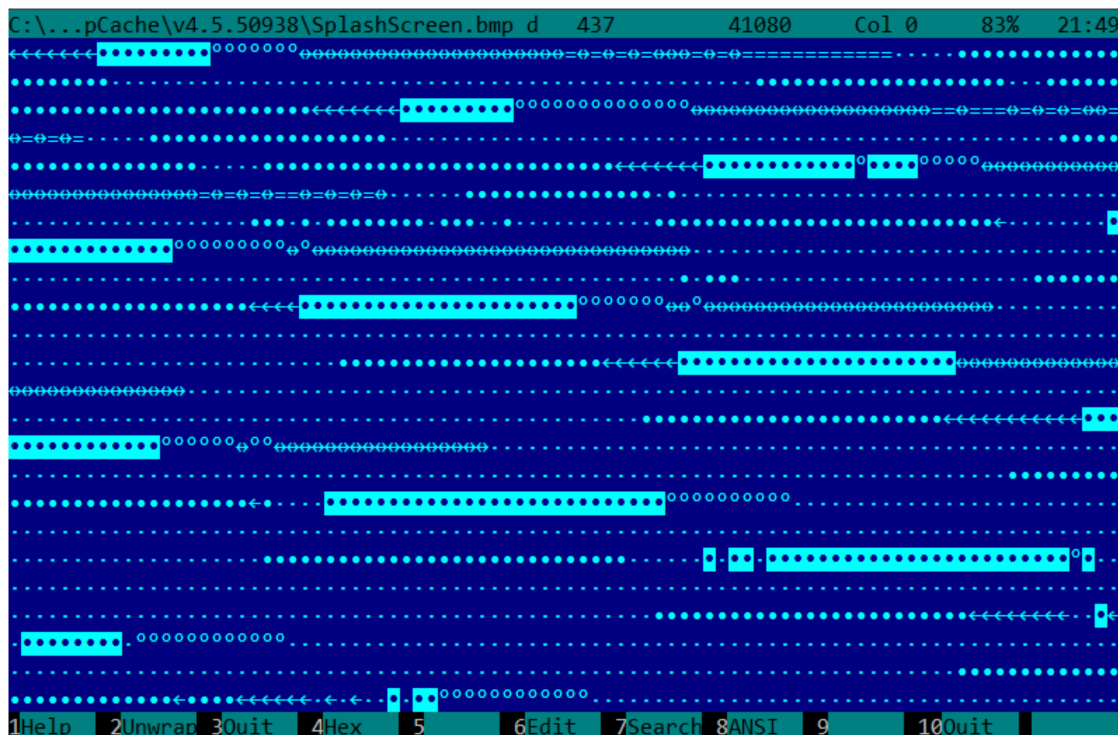


Fig. 5.15: Fragment de fichier BMP

5.10.2 Comparer des «snapshots» mémoire

La technique consistant à comparer directement deux états mémoire afin de voir les changements était souvent utilisée pour tricher avec les jeux sur ordinateurs 8-bit et pour modifier le fichiers des «meilleurs scores».

Par exemple, si vous avez chargé un jeu sur un ordinateur 8-bit (il n'y a pas beaucoup de mémoire dedans, mais le jeu utilise en général encore moins de mémoire), et que vous savez que vous avez maintenant, disons, 100 balles, vous pouvez faire un «snapshot» de toute la mémoire et le sauver quelque part. Puis, vous tirez une fois, le compteur de balles descend à 99, faites un second «snapshot» et puis comparer les deux: il doit y avoir quelque part un octet qui était à 100 au début, et qui est maintenant à 99.

En considérant le fait que ces jeux 8-bit étaient souvent écrits en langage d'assemblage et que de telles variables étaient globales, on peut déterminer avec certitude quelle adresse en mémoire contenait le compteur de balles. Si vous cherchiez toutes les références à cette adresse dans le code du jeu désassemblé, il n'était pas très difficile de trouver un morceau de code décrémentant le compteur de balles, puis d'y écrire une, ou plusieurs, instruction NOP, et d'avoir un jeu avec toujours 100 balles. Les jeux sur ces ordinateurs 8-bit étaient en général chargés à une adresse constante, aussi, il n'y avait pas beaucoup de versions ce chaque jeu (souvent, une seule version était répandue pour un long moment), donc les joueurs enthousiastes

savaient à quelles adresses se trouvaient les octets qui devaient être modifiés (en utilisant l'instruction BASIC POKE) pour le bidouiller. Ceci à conduit à des listes de «cheat» qui contenaient les instructions POKE publiées dans des magazines relatifs aux jeux 8-bit.

De même, il est facile de modifier le fichier des «meilleurs scores», ceci ne fonctionne pas seulement avec des jeux 8-bit. Notez votre score et sauvez le fichier quelque part. Lorsque le décompte des «meilleurs scores» devient différent, comparez juste les deux fichiers, ça peut même être fait avec l'utilitaire DOS FC[25] (les fichiers des «meilleurs scores» sont souvent au format binaire).

Il y aura un endroit où quelques octets seront différents et il est facile de voir lesquels contiennent le score. Toutefois, les développeurs de jeux étaient conscient de ces trucs et pouvaient protéger le programme contre ça.

Exemple quelque peu similaire dans ce livre: 9.3 on page 1241.

Une histoire vraie de 1999

C'était un temps de l'engouement pour la messagerie ICQ, au moins dans les pays de l'ex-URSS. Cette messagerie avait une particularité — certains utilisateurs ne voulaient pas partager leur état en ligne avec tout le monde. Et vous deviez demander une *autorisation* à cet utilisateur. Il pouvait vous autoriser à voir son état, ou pas.

Voici ce que j'ai fait:

- Ajouté un utilisateur.
- Un utiliseur est apparu dans la liste de contact, dans la section “attente d'autorisation”.
- Fermé ICQ.
- Sauvegardé la base de données ICQ.
- Ouvert à nouveau ICQ.
- L'utilisateur m'a *autorisé*.
- Refermé ICQ et comparé les deux base de données.

Il s'est avéré que: les deux bases de données ne différaient que d'un octet. Dans la première version: `RESU\x03`, dans la seconde: `RESU\x02`. (“RESU”, signifie probablement “USER”, i.e., un entête d'une structure où toutes les informations à propos d'un utilisateur étaient stockées.) Cela signifie que l'information sur l'autorisation n'était pas stockée sur le serveur, mais sur le client. Vraisemblablement, la valeur 2/3 reflétait l'état de l'«autorisation».

Registres de Windows

Il est aussi possible de comparer les registres de Windows avant et après l'installation d'un programme.

---

[25]Utilitaire MS-DOS pour comparer des fichiers binaires.

C'est une méthode courante que de trouver quels sont les éléments des registres utilisés par le programme. Peut-être que ceci est la raison pour laquelle le shareware de «nettoyage des registres windows» est si apprécié.

À propos, voici comment sauver les registres de Windows dans des fichiers texte:

```
reg export HKLM HKLM.reg
reg export HKCU HKCU.reg
reg export HKCR HKCR.reg
reg export HKU HKU.reg
reg export HKCC HKCC.reg
```

Ils peuvent être comparés en utilisant diff...

Logiciels d'ingénierie, de CAO, etc.

Si un logiciel utilise des fichiers propriétaires, vous pouvez aussi les examiner. Sauvez un fichier. Puis, ajouter un point ou une ligne ou une autre primitive. Sauvez le fichier, comparez. Ou déplacez un point, sauvez le fichier, comparez.

Comparateur à clignotement

La comparaison de fichiers ou d'images mémoire nous rappelle le comparateur à clignotement [26] : Un dispositif utilisé autrefois par les astronomes pour trouver les objets célestes changeant de position.

Les comparateurs à clignotement permettent d'alterner rapidement entre deux photographies prisent à des moments différents, de façon à faire apparaître les différences visuellement.

À propos, Pluton a été découverte avec un comparateur à clignotement en 1930.

5.11 Détection de l'ISA

Souvent, vous avez à faire à un binaire avec un ISA inconnu. Peut-être que la manière la plus facile de détecter l'ISA est d'en essayer plusieurs dans IDA, objdump ou un autre désassembleur.

Pour réussir ceci, il faut comprendre la différence entre du code incorrectement et celui correctement désassemblé.

5.11.1 Code mal désassemblé

Un rétro ingénieur pratiquant a souvent à faire avec du code mal désassemblé.

[26] https://en.wikipedia.org/wiki/Blink_comparator

Désassemblage depuis une adresse de début incorrecte (x86)

Contrairement à ARM et MIPS (où toute instruction a une longueur de 2 ou 4 octets), les instructions x86 ont une taille variable, donc tout désassembleur démarrant à une mauvaise adresse qui se trouve au milieu d'une instruction x86 pourra produire un résultat incorrect.

À titre d'exemple:

```
add     [ebp-31F7Bh], cl
dec     dword ptr [ecx-3277Bh]
dec     dword ptr [ebp-2CF7Bh]
inc     dword ptr [ebx-7A76F33Ch]
fdiv    st(4), st
db 0FFh
dec     dword ptr [ecx-21F7Bh]
dec     dword ptr [ecx-22373h]
dec     dword ptr [ecx-2276Bh]
dec     dword ptr [ecx-22B63h]
dec     dword ptr [ecx-22F4Bh]
dec     dword ptr [ecx-23343h]
jmp     dword ptr [esi-74h]
xchg    eax, ebp
clc
std
db 0FFh
db 0FFh
mov     word ptr [ebp-214h], cs ; <- le désassembleur a finalement trouvé la
    bonne voie ici
mov     word ptr [ebp-238h], ds
mov     word ptr [ebp-23Ch], es
mov     word ptr [ebp-240h], fs
mov     word ptr [ebp-244h], gs
pushf
pop     dword ptr [ebp-210h]
mov     eax, [ebp+4]
mov     [ebp-218h], eax
lea     eax, [ebp+4]
mov     [ebp-20Ch], eax
mov     dword ptr [ebp-2D0h], 10001h
mov     eax, [eax-4]
mov     [ebp-21Ch], eax
mov     eax, [ebp+0Ch]
mov     [ebp-320h], eax
mov     eax, [ebp+10h]
mov     [ebp-31Ch], eax
mov     eax, [ebp+4]
mov     [ebp-314h], eax
call    ds:IsDebuggerPresent
mov     edi, eax
lea     eax, [ebp-328h]
push    eax
call    sub_407663
pop     ecx
test    eax, eax
```

```
jnz      short loc_402D7B
```

Il y a des instructions incorrectement désassemblées au début, mais finalement le désassembleur revient sur la bonne voie.

À quoi ressemble du bruit aléatoire désassemblé?

Des propriétés répandues qui peuvent être repérées facilement sont:

- Dispersion d'instructions inhabituellement grande. Les instructions x86 les plus fréquentes sont PUSH, MOV, CALL, mais ici nous voyons des instructions de tous les groupes d'instructions: FPU, IN/OUT, instructions systèmes et rares.
- Valeurs grandes et aléatoires, d'offsets et immédiates.
- Sauts ayant des offsets incorrects, sautant au milieu d'autres instructions

Listing 5.7 : bruit aléatoire (x86)

```
    mov     bl, 0Ch
    mov     ecx, 0D38558Dh
    mov     eax, ds:2C869A86h
    db      67h
    mov     dl, 0CCh
    insb
    movsb
    push    eax
    xor     [edx-53h], ah
    fcom    qword ptr [edi-45A0EF72h]
    pop     esp
    pop     ss
    in      eax, dx
    dec     ebx
    push    esp
    lds     esp, [esi-41h]
    retf
    rcl     dword ptr [eax], cl
    mov     cl, 9Ch
    mov     ch, 0DFh
    push    cs
    insb
    mov     esi, 0D9C65E4Dh
    imul    ebp, [ecx], 66h
    pushf
    sal     dword ptr [ebp-64h], cl
    sub     eax, 0AC433D64h
    out     8Ch, eax
    pop     ss
    sbb     [eax], ebx
    aas
    xchg    cl, [ebx+ebx*4+14B31Eh]
    jecxz   short near ptr loc_58+1
    xor     al, 0C6h
    inc     edx
```

```
    db      36h
    pusha
    stosb
    test    [ebx], ebx
    sub     al, 0D3h ; 'L'
    pop     eax
    stosb

loc_58: ; CODE XREF: seg000:0000004A
    test    [esi], eax
    inc     ebp
    das
    db      64h
    pop     ecx
    das
    hlt

    pop     edx
    out     0B0h, al
    lodsb
    push    ebx
    cdq
    out     dx, al
    sub     al, 0Ah
    sti
    outsd
    add     dword ptr [edx], 96FCBE4Bh
    and     eax, 0E537EE4Fh
    inc     esp
    stosd
    cdq
    push    ecx
    in      al, 0CBh
    mov     ds:0D114C45Ch, al
    mov     esi, 659D1985h
```

Listing 5.8 : bruit aléatoire (x86-64)

```
    lea     esi, [rax+rdx*4+43558D29h]

loc_AF3: ; CODE XREF: seg000:0000000000000B46
    rcl     byte ptr [rsi+rax*8+29BB423Ah], 1
    lea     ecx, cs:0FFFFFFFFB2A6780Fh
    mov     al, 96h
    mov     ah, 0CEh
    push    rsp
    lods    byte ptr [esi]

    db  2Fh ; /

    pop     rsp
    db      64h
    retf    0E993h
```

```
    cmp     ah, [rax+4Ah]
    movzx   rsi, dword ptr [rbp-25h]
    push    4Ah
    movzx   rdi, dword ptr [rdi+rdx*8]

    db  9Ah

    rcr     byte ptr [rax+1Dh], cl
    lodsd
    xor     [rbp+6CF20173h], edx
    xor     [rbp+66F8B593h], edx
    push    rbx
    sbb     ch, [rbx-0Fh]
    stosd
    int     87h
    db      46h, 4Ch
    out     33h, rax
    xchg    eax, ebp
    test    ecx, ebp
    movsd
    leave
    push    rsp

    db  16h

    xchg    eax, esi
    pop     rdi

loc_B3D: ; CODE XREF: seg000:0000000000000B5F
    mov     ds:93CA685DF98A90F9h, eax
    jnz     short near ptr loc_AF3+6
    out     dx, eax
    cwde
    mov     bh, 5Dh ; ']'
    movsb
    pop     rbp
```

Listing 5.9 : bruit aléatoire (ARM (Mode ARM))

```
BLNE    0xFE16A9D8
BGE     0x1634D0C
SVCCS   0x450685
STRNVT  R5, [PC],#-0x964
LDCGE   p6, c14, [R0],#0x168
STCCSL  p9, c9, [LR],#0x14C
CMNHIP  PC, R10,LSL#22
FLDMIADNV LR!, {D4}
MCR     p5, 2, R2,c15,c6, 4
BLGE    0x1139558
BLGT    0xFF9146E4
STRNEB  R5, [R4],#0xCA2
STMNEIB R5, {R0,R4,R6,R7,R9-SP,PC}
STMIA   R8, {R0,R2-R4,R7,R8,R10,SP,LR}^
STRB    SP, [R8],PC,ROR#18
```

```
LDCCS   p9, c13, [R6,#0x1BC]
LDRGE   R8, [R9,#0x66E]
STRNEB  R5, [R8],#-0x8C3
STCCSL  p15, c9, [R7,#-0x84]
RSBLS   LR, R2, R11,ASR LR
SVCGT   0x9B0362
SVCGT   0xA73173
STMNEDB R11!, {R0,R1,R4-R6,R8,R10,R11,SP}
STR     R0, [R3],#-0xCE4
LDCGT   p15, c8, [R1,#0x2CC]
LDRCCB  R1, [R11],-R7,ROR#30
BLLT    0xFED9D58C
BL      0x13E60F4
LDMVSIB R3!, {R1,R4-R7}^
USATNE  R10, #7, SP,LSL#11
LDRGEB  LR, [R1],#0xE56
STRPLT  R9, [LR],#0x567
LDRLT   R11, [R1],#-0x29B
SVCNV   0x12DB29
MVNNVS  R5, SP,LSL#25
LDCL    p8, c14, [R12,#-0x288]
STCNEL  p2, c6, [R6,#-0xBC]!
SVCNV   0x2E5A2F
BLX     0x1A8C97E
TEQGE   R3, #0x1100000
STMLSIA R6, {R3,R6,R10,R11,SP}
BICPLS  R12, R2, #0x5800
BNE     0x7CC408
TEQGE   R2, R4,LSL#20
SUBS    R1, R11, #0x28C
BICVS   R3, R12, R7,ASR R0
LDRMI   R7, [LR],R3,LSL#21
BLMI    0x1A79234
STMVCDB R6, {R0-R3,R6,R7,R10,R11}
EORMI   R12, R6, #0xC5
MCRRCS  p1, 0xF, R1,R3,c2
```

Listing 5.10 : bruit aléatoire (ARM (Mode Thumb))

```
    LSRS    R3, R6, #0x12
    LDRH    R1, [R7,#0x2C]
    SUBS    R0, #0x55 ; 'U'
    ADR     R1, loc_3C
    LDR     R2, [SP,#0x218]
    CMP     R4, #0x86
    SXTB    R7, R4
    LDR     R4, [R1,#0x4C]
    STR     R4, [R4,R2]
    STR     R0, [R6,#0x20]
    BGT     0xFFFFFF72
    LDRH    R7, [R2,#0x34]
    LDRSH   R0, [R2,R4]
    LDRB    R2, [R7,R2]
```

```
    DCB 0x17
    DCB 0xED

    STRB    R3, [R1,R1]
    STR     R5, [R0,#0x6C]
    LDMIA   R3, {R0-R5,R7}
    ASRS    R3, R2, #3
    LDR     R4, [SP,#0x2C4]
    SVC     0xB5
    LDR     R6, [R1,#0x40]
    LDR     R5, =0xB2C5CA32
    STMIA   R6, {R1-R4,R6}
    LDR     R1, [R3,#0x3C]
    STR     R1, [R5,#0x60]
    BCC     0xFFFFFF70
    LDR     R4, [SP,#0x1D4]
    STR     R5, [R5,#0x40]
    ORRS    R5, R7

loc_3C ; DATA XREF: ROM:00000006
    B       0xFFFFFF98
```

Listing 5.11 : bruit aléatoire (MIPS little endian)

```
    lw      $t9, 0xCB3($t5)
    sb      $t5, 0x3855($t0)
    sltiu   $a2, $a0, -0x657A
    ldr     $t4, -0x4D99($a2)
    daddi   $s0, $s1, 0x50A4
    lw      $s7, -0x2353($s4)
    bgtzl   $a1, 0x17C5C

    .byte 0x17
    .byte 0xED
    .byte 0x4B  # K
    .byte 0x54  # T

    lwc2    $31, 0x66C5($sp)
    lwu     $s1, 0x10D3($a1)
    ldr     $t6, -0x204B($zero)
    lwc1    $f30, 0x4DBE($s2)
    daddiu  $t1, $s1, 0x6BD9
    lwu     $s5, -0x2C64($v1)
    cop0    0x13D642D
    bne     $gp, $t4, 0xFFFF9EF0
    lh      $ra, 0x1819($s1)
    sdl     $fp, -0x6474($t8)
    jal     0x78C0050
    ori     $v0, $s2, 0xC634
    blez    $gp, 0xFFFEA9D4
    swl     $t8, -0x2CD4($s2)
    sltiu   $a1, $k0, 0x685
    sdc1    $f15, 0x5964($at)
    sw      $s0, -0x19A6($a1)
```

```
    sltiu   $t6, $a3, -0x66AD
    lb      $t7, -0x4F6($t3)
    sd      $fp, 0x4B02($a1)
```

Il est important de garder à l'esprit que du code de dépaquetage et de déchiffrement construit intelligemment (y compris auto-modifiant) peut avoir l'air aléatoire, mais s'exécute toujours correctement.

5.11.2 Code désassemblé correctement

Chaque ISA a une douzaine d'instructions les plus utilisées, toutes les autres le sont beaucoup moins souvent.

Concernant le x86, il est intéressant de savoir le fait que les instructions d'appel de fonctions (PUSH/CALL/ADD) et MOV sont les morceaux de code les plus fréquemment exécutées dans presque tous les programmes que nous utilisons. Autrement dit, le CPU est très occupé à passer de l'information entre les niveaux d'abstraction, ou, on peut dire qu'il est très occupé à commuter entre ces niveaux. Indépendamment du type d'ISA. Ceci a un coût de diviser les problèmes entre plusieurs niveaux d'abstraction (ainsi les êtres humain peuvent travailler plus facilement avec).

5.12 Autres choses

5.12.1 Idée générale

Un rétro-ingénieur doit essayer se se mettre dans la peau d'un programmeur aussi souvent que possible. Pour adopter son point de vue et se demander comment il aurait résolu des tâches d'un cas spécifique.

5.12.2 Ordre des fonctions dans le code binaire

Toutes les fonctions situées dans un unique fichier .c ou .cpp sont compilées dans le fichier objet (.o) correspondant. Plus tard, l'éditeur de liens mets tous les fichiers dont il a besoin ensemble, sans changer l'ordre ni les fonctions. Par conséquent, si vous voyez deux ou plus fonctions consécutives, cela signifie qu'elles étaient situées dans le même fichier source (à moins que vous ne soyez en limite de deux fichiers objet, bien sûr). Ceci signifie que ces fonctions ont quelque chose en commun, qu'elles sont des fonctions du même niveau d'API, de la même bibliothèque, etc.

Ceci est une histoire vraie de pratique: il était une fois, alors que je cherchais des fonctions relatives à Twofish dans un programme lié à la bibliothèque CryptoPP, en particulier des fonctions de chiffrement/déchiffrement.
J'ai trouvé la fonction `Twofish::Base::UncheckedSetKey()` mais pas d'autres. Après avoir cherché dans le code source `twofish.cpp`[27], il devint clair que toutes les fonctions étaient situées dans ce module (`twofish.cpp`).

[27] https://github.com/weidai11/cryptopp/blob/b613522794a7633aa2bd81932a98a0b0a51bc04f/twofish.cpp

Donc j'ai essayé toutes les fonctions qui suivaient `Twofish::Base::UncheckedSetKey()`—comme elles arrivaient,
une a été `Twofish::Enc::ProcessAndXorBlock()`,
une autre—`Twofish::Dec::ProcessAndXorBlock()`.

5.12.3 Fonctions minuscules

Les fonctions minuscules comme les fonctions vides (1.3 on page 8) ou les fonctions qui renvoient juste “true” (1) ou “false” (0) (1.4 on page 10) sont très communes, et presque tous les compilateurs corrects tendent à ne mettre qu'une seule fonction de ce genre dans le code de l'exécutable résultant, même si il y avait plusieurs fonctions similaires dans le code source. Donc, à chaque fois que vous voyez une fonction minuscule consistant seulement en `mov eax, 1 / ret` qui est référencée (et peut être appelée) dans plusieurs endroits qui ne semblent pas reliés les uns au autres, ceci peut résulter d'une telle optimisation.

5.12.4 C++

Les données RTTI (3.21.1 on page 720)- peuvent être utiles pour l'identification des classes C++.

5.12.5 Crash délibéré

Souvent, vous voulez savoir quelle fonction a été exécutée, et laquelle ne l'a pas été. Vous pouvez utiliser un débogueur, mais sur des architectures exotiques, il peut ne pas en avoir, donc la façon la plus simple est d'y mettre un opcode invalide, ou quelque chose comme `INT3` (0xCC). Le crash signalera le fait que l'instruction a été exécutée.

Un autre exemple de crash délibéré: 3.23.4 on page 787.

Chapitre 6

Spécifique aux OS

6.1 Méthodes de transmission d'arguments (calling conventions)

6.1.1 cdecl

Il s'agit de la méthode la plus courante pour passer des arguments à une fonction en langage C/C++.

Après que l'appelé ait rendu la main, l'appelant doit ajuster la valeur du pointeur de pile (ESP) pour lui redonner la valeur qu'elle avait avant l'appel de l'appelé.

Listing 6.1 : cdecl

```
push arg3
push arg2
push arg1
call function
add esp, 12 ; returns ESP
```

6.1.2 stdcall

Similaire à *cdecl*, sauf que c'est l'appelé qui doit réinitialise ESP à sa valeur d'origine en utilisant l'instruction RET x et non pas RET,
avec `x = nb arguments * sizeof(int)`[1]. Après le retour du callee, l'appelant ne modifie pas le pointeur de pile. Il ny' a donc pas d'instruction `add esp, x`.

Listing 6.2 : stdcall

```
push arg3
push arg2
push arg1
call function
```

[1]La taille d'une variable de type *int* est de 4 octets sur les systèmes x86 et de 8 octets sur les systèmes x64

```
function:
;... do something ...
ret 12
```

Cette méthode est omniprésente dans les librairies standard win32, mais pas dans celles win64 (voir ci-dessous pour win64).

Prenons par exemple la fonction 1.89 on page 132 et modifions la légèrement en utilisant la convention `__stdcall` :

```
int __stdcall f2 (int a, int b, int c)
{
        return a*b+c;
};
```

Le résultat de la compilation sera quasiment identique à celui de 1.90 on page 132, mais vous constatez la présence de RET 12 au lieu de RET. L' appelant ne met pas à jour SP après l'appel.

Avec la convention `__stdcall`, on peut facilement déduire le nombre d'arguments de la fonction appelée à partir de l'instruction RETN n : divisez n par 4.

Listing 6.3 : MSVC 2010

```
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_c$ = 16        ; size = 4
_f2@12  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     12
_f2@12  ENDP

; ...
        push    3
        push    2
        push    1
        call    _f2@12
        push    eax
        push    OFFSET $SG81369
        call    _printf
        add     esp, 8
```

Fonctions à nombre d'arguments variables

Les fonctions du style `printf()` sont un des rares cas de fonctions à nombre d'arguments variables en C/C++. Elles permettent d'illustrer une différence importante

entre les conventions *cdecl* et *stdcall*. Partons du principe que le compilateur connait le nombre d'arguments utilisés à chaque fois que la fonction `printf()` est appelée.

En revanche, la fonction `printf()` est déjà compilée dans MSVCRT.DLL (si l'on parle de Windows) et ne possède aucune information sur le nombre d'arguments qu'elle va recevoir. Elle peut cependant le deviner à partir du contenu du paramètre format.

Si la convention *stdcall* était utilisé pour la fonction `printf()`, elle devrait réajuster le pointeur de pile à sa valeur initiale en comptant le nombre d'arguments dans la chaîne de format. Cette situation serait dangereuse et pourrait provoquer un crash du programme à la moindre faute de frappe du programmeur. La convention *stdcall* n'est donc pas adaptée à ce type de fonction. La convention *cdecl* est préférable.

6.1.3 fastcall

Il s'agit d'un nom générique pour désigner les conventions qui passent une partie des paramètres dans des registres et le reste sur la pile. Historiquement, cette méthode était plus performante que les conventions *cdecl*/*stdcall* - car elle met moins de pression sur la pile. Ce n'est cependant probablement plus le cas sur les processeurs actuels qui sont beaucoup plus complexes.

Cette convention n'est pas standardisée. Les compilateurs peuvent donc l'implémenter à leur guise. Prenez une DLL qui en utilise une autre compilée avec une interprétation différente de la convention *fastcall*. Vous avez un cas d'école et des problèmes en perspective.

Les compilateurs MSVC et GCC passent les deux premiers arguments dans ECX et EDX, et le reste des arguments sur la pile.

Le pointeur de pile doit être restauré à sa valeur initiale par l'appelé (comme pour la convention *stdcall*).

Listing 6.4 : fastcall

```
push arg3
mov edx, arg2
mov ecx, arg1
call function

function:
.. do something ..
ret 4
```

Prenons par exemple la fonction 1.89 on page 132 et modifions la légèrement en utilisant la convention `__fastcall` :

```
int __fastcall f3 (int a, int b, int c)
{
        return a*b+c;
};
```

Le résultat de la compilation est le suivant:

Listing 6.5 : MSVC 2010 optimisé/Ob0

```
_c$ = 8         ; size = 4
@f3@12  PROC
; _a$ = ecx
; _b$ = edx
        mov     eax, ecx
        imul    eax, edx
        add     eax, DWORD PTR _c$[esp-4]
        ret     4
@f3@12  ENDP

; ...

        mov     edx, 2
        push    3
        lea     ecx, DWORD PTR [edx-1]
        call    @f3@12
        push    eax
        push    OFFSET $SG81390
        call    _printf
        add     esp, 8
```

Nous voyons que l'appelé ajuste SP en utilisant l'instruction RETN suivie d'un opérande.

Le nombre d'arguments peut, encore une fois, être facilement déduit.

GCC regparm

Il s'agit en quelque sorte d'une évolution de la convention *fastcall*[2]. L'option `-mregparm` permet de définir le nombre d'arguments (3 au maximum) qui doivent être passés dans les registres. EAX, EDX et ECX sont alors utilisés.

Bien entendu si le nombre d'arguments est inférieur à 3, seuls les premiers registres sont utilisés.

C'est l'appelant qui effectue l'ajustement du pointeur de pile.

Pour un exemple, voire (1.28.1 on page 395).

Watcom/OpenWatcom

Dans le cas de ce compilateur, on parle de «register calling convention». Les 4 premiers arguments sont passés dans les registres EAX, EDX, EBX et ECX. Les paramètres suivant sont passés sur la pile.

Un suffixe constitué d'un caractère de soulignement est ajouté par le compilateur au nom de la fonction afin de les distinguer de celles qui utilisent une autre convention d'appel.

[2] http://www.ohse.de/uwe/articles/gcc-attributes.html#func-regparm

6.1.4 thiscall

Cette convention passe le pointeur d'objet *this* à une méthode en C++.

Le compilateur MSVC, passe généralement le pointeur *this* dans le registre ECX.

Le compilateur GCC passe le pointeur *this* comme le premier argument de la fonction. Thus it will be very visible that internally: all function-methods have an extra argument.

Pour un example, voir (3.21.1 on page 699).

6.1.5 x86-64

Windows x64

La méthode utilisée pour passer les arguments aux fonctions en environnement Win64 ressemble beaucoup à la convention `fastcall`. Les 4 premiers arguments sont passés dans les registres RCX, RDX, R8 et R9, les arguments suivants sont passés sur la pile. L'appelant doit également préparer un espace sur la pile pour 32 octets, soit 4 mots de 64 bits, l'appelé pourra y sauvegarder les 4 premiers arguments. Les fonctions suffisamment simples peuvent utiliser les paramètres directement depuis les registres. Les fonctions plus complexes doivent sauvegarder les valeurs de paramètres afin de les utiliser plus tard.

L'appelé est aussi responsable de rétablir la valeur du pointeur de pile à la valeur qu'il avait avant l'appel de la fonction.

Cette convention est utilisée dans les DLLs Windows x86-64 (à la place de *stdcall* en win32).

Exemple:

```
#include <stdio.h>

void f1(int a, int b, int c, int d, int e, int f, int g)
{
        printf ("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
};

int main()
{
        f1(1,2,3,4,5,6,7);
};
```

Listing 6.6 : MSVC 2012 /0b

```
$SG2937 DB      '%d %d %d %d %d %d %d', 0aH, 00H

main    PROC
        sub     rsp, 72

        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        mov     DWORD PTR [rsp+32], 5
```

```
        mov     r9d, 4
        mov     r8d, 3
        mov     edx, 2
        mov     ecx, 1
        call    f1

        xor     eax, eax
        add     rsp, 72
        ret     0
main    ENDP

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1      PROC
$LN3:
        mov     DWORD PTR [rsp+32], r9d
        mov     DWORD PTR [rsp+24], r8d
        mov     DWORD PTR [rsp+16], edx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 72

        mov     eax, DWORD PTR g$[rsp]
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
        mov     eax, DWORD PTR d$[rsp]
        mov     DWORD PTR [rsp+32], eax
        mov     r9d, DWORD PTR c$[rsp]
        mov     r8d, DWORD PTR b$[rsp]
        mov     edx, DWORD PTR a$[rsp]
        lea     rcx, OFFSET FLAT:$SG2937
        call    printf

        add     rsp, 72
        ret     0
f1      ENDP
```

Nous vyons ici clairement que des 7 arguments, 4 sont passés dans les registres et les 3 suivants sur la pile.

Le prologue du code de la fonction f1() sauvegarde les arguments dans le «scratch space»—un espace sur la pile précisément prévu à cet effet.

Le compilateur agit ainsi car il n'est pas certain par avance qu'il disposera de suffisamment de registres pour toute la fonction en l'absence des 4 utilisés par les paramètres.

L'appelant est responsable de l'allocation du «scratch space» sur la pile.

Listing 6.7 : avec optimisation MSVC 2012 /0b

```
$SG2777 DB      '%d %d %d %d %d %d %d', 0aH, 00H

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1      PROC
$LN3:
        sub     rsp, 72

        mov     eax, DWORD PTR g$[rsp]
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
        mov     DWORD PTR [rsp+32], r9d
        mov     r9d, r8d
        mov     r8d, edx
        mov     edx, ecx
        lea     rcx, OFFSET FLAT:$SG2777
        call    printf

        add     rsp, 72
        ret     0
f1      ENDP

main    PROC
        sub     rsp, 72

        mov     edx, 2
        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        lea     r9d, QWORD PTR [rdx+2]
        lea     r8d, QWORD PTR [rdx+1]
        lea     ecx, QWORD PTR [rdx-1]
        mov     DWORD PTR [rsp+32], 5
        call    f1

        xor     eax, eax
        add     rsp, 72
        ret     0
main    ENDP
```

L'exemple compilé en branchant les optimisations, sera quasiment identique, si ce n'est que le «scratch space» ne sera pas utilisé car inutile.

Notez également la manière dont MSVC 2012 optimise le chargement de certaines valeurs litérales dans les registres en utilisant LEA (.1.6 on page 1333). L'instruction MOV utiliserait 1 octet de plus (5 au lieu de 4).

8.2.1 on page 1042 est un autre exemple de cette pratique.

Windows x64: Passage de *this* (C/C++)

Le pointeur *this* est passé dans le registre RCX, le premier argument de la méthode dans RDX, etc. Pour un exemple, voir : 3.21.1 on page 702.

Linux x64

Le passage d'arguments par Linux pour x86-64 est quasiment identique à celui de Windows, si ce n'est que 6 registres sont utilisés au lieu de 4 (RDI, RSI, RDX, RCX, R8, R9) et qu'il n'existe pas de «scratch space». L'glslinkcalleeappelé conserve la possibilité de sauvegarder les registres sur la pile s'il le souhaite ou en a besoin.

Listing 6.8 : GCC 4.7.3 avec optimisation

```
.LC0:
        .string "%d %d %d %d %d %d %d\n"
f1:
        sub     rsp, 40
        mov     eax, DWORD PTR [rsp+48]
        mov     DWORD PTR [rsp+8], r9d
        mov     r9d, ecx
        mov     DWORD PTR [rsp], r8d
        mov     ecx, esi
        mov     r8d, edx
        mov     esi, OFFSET FLAT:.LC0
        mov     edx, edi
        mov     edi, 1
        mov     DWORD PTR [rsp+16], eax
        xor     eax, eax
        call    __printf_chk
        add     rsp, 40
        ret
main:
        sub     rsp, 24
        mov     r9d, 6
        mov     r8d, 5
        mov     DWORD PTR [rsp], 7
        mov     ecx, 4
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f1
        add     rsp, 24
        ret
```

N.B.: Les valeurs sont enregistrées dans la partie basse des registres (e.g., EAX) et non pas dans la totalité du registre 64 bits (RAX). Ceci s'explique par le fait que l'écriture des 32 bits de poids faible du registre remet automatiquement à zéro les 32 bits de poids fort. On suppose qu'AMD a pris cette décision afin de simplifier le portage du code 32 bits vers l'architecture x86-64.

6.1.6 Valeur de retour de type *float* et *double*

Dans toutes les conventions, à l'exception de l'environnement Win64, les valeurs de type *float* ou *double* sont retournées dans le registre FPU `ST(0)`.

En environnement Win64, les valeurs de type *float* et *double* sont respectivement retournées dans les 32 bits de poids faible et dans les 64 bits du registre `XMM0`.

6.1.7 Modification des arguments

Il arrive que les programmeurs C/C++ (et ceux d'autres LPs aussi) se demandent ce qui se passe s'ils modifient la valeur des arguments dans la fonction appelée.

La réponse est simple. Les arguments sont stockés sur la pile, et c'est elle qui est modifiée.

Une fois que l'appelé s'achève, la fonction appelante ne les utilise pas. (Je n'a jamais constaté qu'il en aille autrement).

```
#include <stdio.h>

void f(int a, int b)
{
        a=a+b;
        printf ("%d\n", a);
};
```

Listing 6.9 : MSVC 2012

```
_a$ = 8                                                 ; size = 4
_b$ = 12                                                ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
        mov     DWORD PTR _a$[ebp], eax
        mov     ecx, DWORD PTR _a$[ebp]
        push    ecx
        push    OFFSET $SG2938 ; '%d', 0aH
        call    _printf
        add     esp, 8
        pop     ebp
        ret     0
_f      ENDP
```

Donc, oui, la valeur des arguments peut être modifiée sans problème. Sous réserver que l'argument ne soit pas une *references* en C++ (3.21.3 on page 721), et que vous ne modifiez pas la valeur qui est référencée par un pointeur, l'effet de votre modification ne se propagera pas au dehors de la fonction.

En théorie, après le retour de l'appelé, l'appelant pourrait récupérer l'argument modifié et l'utiliser à sa guise. Ceci pourrait peut être se faire dans un programme rédigé en assembleur.

Par exemple, un compilateur C/C++ génèrera un code comme celui-ci :

```
        push    456     ; will be b
        push    123     ; will be a
        call    f       ; f() modifies its first argument
        add     esp, 2*4
```

Nous pouvons réécrire le code ainsi :

```
        push    456     ; will be b
        push    123     ; will be a
        call    f       ; f() modifies its first argument
        pop     eax
        add     esp, 4
        ; EAX=1st argument of f() modified in f()
```

Il est difficile d'imaginer pourquoi quelqu'un aurait besoin d'agir ainsi, mais en pratique c'est possible. Toujours est-il que les langages C/C++ ne permettent pas de faire ainsi.

6.1.8 Recevoir un argument par adresse

…mieux que cela, il est possible de passer à une fonction l'adresse d'un argument, plutôt que la valeur de l'argument lui-même:

```
#include <stdio.h>

// located in some other file
void modify_a (int *a);

void f (int a)
{
        modify_a (&a);
        printf ("%d\n", a);
};
```

Il est difficile de comprendre ce fonctionnement jusqu'à ce que l'on regarde le code:

Listing 6.10 : MSVC 2010 optimisé

```
$SG2796 DB      '%d', 0aH, 00H

_a$ = 8
_f      PROC
```

```
        lea     eax, DWORD PTR _a$[esp-4] ; just get the address of value in
    local stack
        push    eax                       ; and pass it to modify_a()
        call    _modify_a
        mov     ecx, DWORD PTR _a$[esp]   ; reload it from the local stack
        push    ecx                       ; and pass it to printf()
        push    OFFSET $SG2796            ; '%d'
        call    _printf
        add     esp, 12
        ret     0
_f      ENDP
```

L'argument a est placé sur la pile et l'adresse de cet emplacement de pile est passé à une autre fonction. Celle-ci modifie la valeur à l'adresse référencée par le pointeur, puis `printf()` affiche la valeur après modification.

Le lecteur attentif se demandera peut-être ce qu'il en est avec les conventions d'appel qui utilisent les registres pour passer les arguments.

C'est justement une des utilisations du *Shadow Space*.

La valeur en entrée est copiée du registre vers le *Shadow Space* dans la pile locale, puis l'adresse de la pile locale est passée à la fonction appelée:

Listing 6.11 : MSVC 2012 x64 optimisé

```
$SG2994 DB      '%d', 0aH, 00H

a$ = 48
f       PROC
        mov     DWORD PTR [rsp+8], ecx   ; save input value in Shadow Space
        sub     rsp, 40
        lea     rcx, QWORD PTR a$[rsp]   ; get address of value and pass it
    to modify_a()
        call    modify_a
        mov     edx, DWORD PTR a$[rsp]   ; reload value from Shadow Space and
    pass it to printf()
        lea     rcx, OFFSET FLAT:$SG2994 ; '%d'
        call    printf
        add     rsp, 40
        ret     0
f       ENDP
```

Le compilateur GCC lui aussi sauvegarde la valeur sur la pile locale:

Listing 6.12 : GCC 4.9.1 optimisé x64

```
.LC0:
        .string "%d\n"
f:
        sub     rsp, 24
        mov     DWORD PTR [rsp+12], edi  ; store input value to the local
    stack
        lea     rdi, [rsp+12]            ; take an address of the value and
    pass it to modify_a()
        call    modify_a
```

```
        mov     edx, DWORD PTR [rsp+12] ; reload value from the local stack
    and pass it to printf()
        mov     esi, OFFSET FLAT:.LC0    ; '%d'
        mov     edi, 1
        xor     eax, eax
        call    __printf_chk
        add     rsp, 24
        ret
```

Le compilateur GCC pour ARM64 se comporte de la même manière, mais l'espace de sauvegarde sur la pile est dénommé *Register Save Area* :

Listing 6.13 : GCC 4.9.1 optimisé ARM64

```
f:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0          ; setup FP
        add     x1, x29, 32         ; calculate address of variable in
    Register Save Area
        str     w0, [x1,-4]!        ; store input value there
        mov     x0, x1              ; pass address of variable to the
    modify_a()
        bl      modify_a
        ldr     w1, [x29,28]        ; load value from the variable and pass it
    to printf()
        adrp    x0, .LC0            ; '%d'
        add     x0, x0, :lo12:.LC0
        bl      printf              ; call printf()
        ldp     x29, x30, [sp], 32
        ret
.LC0:
        .string "%d\n"
```

Enfin, nous constatons un usage similaire du *Shadow Space* ici: 3.17.1 on page 668.

6.1.9 Problème des ctypes en Python (devoir à la maison en assembleur x86)

Un module Python ctypes peut appeler des fonctions externes dans des DLLs, .so's, etc. Mais la convention d'appel (pour l'environnement 32-bit) doit être définie explicitement:

```
"ctypes" exports the *cdll*, and on Windows *windll* and *oledll*
objects, for loading dynamic link libraries.

You load libraries by accessing them as attributes of these objects.
*cdll* loads libraries which export functions using the standard
"cdecl" calling convention, while *windll* libraries call functions
using the "stdcall" calling convention.
```

(https://docs.python.org/3/library/ctypes.html)[3]

[3] NDT:Projet de traduction en français: https://docs.python.org/fr/3/library/ctypes.html

En fait, nous pouvons modifier le module ctypes (ou tout autre module d'appel), afin qu'il appelle avec succès des fonctions externes cdecl ou stdcall, sans connaître, ce qui se trouve où. (Le nombre d'arguments, toutefois, doit être spécifié).

Il est possible de le résoudre en utilisant environ 5 à 10 instructions assembleur x86 dans l'appelant. Essayez de trouver ça.

6.1.10 Exemple cdecl: DLL

Revenons au fait que la façon de déclarer la fonction `main()` n'est pas très importante: 1.9.2 on page 46.

Ceci est une histoire vraie: il était une fois où je voulais remplacer un fichier original de DLL par le mien. Premièrement, j'ai énuméré les noms de tous les exports de la DLL et j'ai écrit une fonction dans ma propre DLL de remplacement pour chaque fonction de la DLL originale, comme:

```
void function1 ()
{
        write_to_log ("function1() called\n");
};
```

Je voulais voir quelles fonctions étaient appelées lors de l'exécution, et quand. Toutefois, j'étais pressé et n'avais pas le temps de déduire le nombre d'arguments pour chaque fonction, encore moins pour les types des données. Donc chaque fonction dans ma DLL de remplacement n'avait aucun argument que ce soit. Mais tout fonctionnait, car toutes les fonctions utilisaient la convention d'appel *cdecl*. (Ça ne fonctionnerait pas si les fonctions avaient la convention d'appel *stdcall*.) Ça a aussi fonctionné pour la version x64.

Et puis j'ai fait une étape de plus: j'ai déduit les types des arguments pour certaines fonctions. Mais j'ai fait quelques erreurs, par exemple, la fonction originale prend 3 arguments, mais je n'en ai découvert que 2, etc.

Cependant, ça fonctionnait. Au début, ma DLL de remplacement ignorait simplement tous les arguments. Puis, elle ignorait le 3ème argument.

6.2 Thread Local Storage

TLS est un espace de données propre à chaque thread, qui peut y conserver ce qu'il estime utile. Un exemple d'utilisation bien connu en est la variable globale *errno* du standard C.

Plusieurs threads peuvent invoquer simultanément différentes fonctions qui retournent toutes un code erreur dans la variable *errno*. L'utilisation d'une variable globale dans le contexte d'un programme comportant plusieurs threads serait donc inadaptée dans ce cas. C'est pourquoi la variable *errno* est conservée dans l'espace TLS.

La version 11 du standard C++ a ajouté un nouveau modificateur *thread_local* qui

indique que chaque thread possède sa propre copie de la variable, qui peut-être initialisée et est alors conservée dans l'espace TLS [4] :

Listing 6.14 : C++11

```
#include <iostream>
#include <thread>

thread_local int tmp=3;

int main()
{
        std::cout << tmp << std::endl;
};
```

Compilé avec MinGW GCC 4.8.1, mais pas avec MSVC 2012.

Dans le contexte des fichiers au format PE, la variable *tmp* sera allouée dans la section dédiée au TLS du fichier exécutable résultant de la compilation.

6.2.1 Amélioration du générateur linéaire congruent

Le générateur de nombres pseudo-aléatoires 1.29 on page 436 que nous avons étudié précédemment contient une faiblesse. Son comportement est buggé dans un environnement multi-thread. Il utilise en effet une variable d'état dont la valeur peut être lue et/ou modifiée par plusieurs threads simultanément.

Win32

Données TLS non initialisées

Une solution consiste à déclarer la variable globale avec le modificateur `__declspec( thread )`. Elle sera alors allouée dans le TLS (ligne 9) :

```
1  #include <stdint.h>
2  #include <windows.h>
3  #include <winnt.h>
4
5  // from the Numerical Recipes book:
6  #define RNG_a 1664525
7  #define RNG_c 1013904223
8
9  __declspec( thread ) uint32_t rand_state;
10
11 void my_srand (uint32_t init)
12 {
13         rand_state=init;
14 }
15
16 int my_rand ()
```

[4] C11 supporte aussi les thread, mais optionel

```
{
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
}

int main()
{
        my_srand(0x12345678);
        printf ("%d\n", my_rand());
};
```

Hiew nous montre alors qu'il existe une nouvelle section nommée `.tls` dans le fichier PE.

Listing 6.15 : avec optimisation MSVC 2013 x86

```
_TLS    SEGMENT
_rand_state DD  01H DUP (?)
_TLS    ENDS

_DATA   SEGMENT
$SG84851 DB     '%d', 0aH, 00H
_DATA   ENDS
_TEXT   SEGMENT

_init$ = 8      ; size = 4
_my_srand PROC
; FS:0=address of TIB
        mov     eax, DWORD PTR fs:__tls_array  ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
        mov     ecx, DWORD PTR __tls_index
        mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
        mov     eax, DWORD PTR _init$[esp-4]
        mov     DWORD PTR _rand_state[ecx], eax
        ret     0
_my_srand ENDP

_my_rand PROC
; FS:0=address of TIB
        mov     eax, DWORD PTR fs:__tls_array  ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
        mov     ecx, DWORD PTR __tls_index
        mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
        imul    eax, DWORD PTR _rand_state[ecx], 1664525
        add     eax, 1013904223         ; 3c6ef35fH
        mov     DWORD PTR _rand_state[ecx], eax
        and     eax, 32767              ; 00007fffH
        ret     0
_my_rand ENDP

_TEXT   ENDS
```

La variable rand_state se trouve donc maintenant dans le segment TLS et chaque thread en possède sa propre version de cette variable.

Voici comment elle est accédée. L'adresse du TIB est chargée depuis FS:2Ch, un index est ajouté (si nécessaire), puis l'adresse du segment TLS est calculée.

Il est ainsi possible d'accéder la valeur de la variable rand_state au travers du registre ECX qui contient une adresse propre à chaque thread.

Le sélecteur FS: est connu de tous les rétro-ingénieur. Il est spécifiquement utilisé pour toujours contenir l'adresse du TIB du thread en cours d'exécution. L'accès aux données propres à chaque thread est donc une opération performante.

En environnement Win64, c'est le sélecteur GS: qui est utilisé pour ce faire. L'adresse de l'espace TLS y est conservé à l'offset 0x58 :

Listing 6.16 : avec optimisation MSVC 2013 x64

```
_TLS    SEGMENT
rand_state DD   01H DUP (?)
_TLS    ENDS

_DATA   SEGMENT
$SG85451 DB     '%d', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT

init$ = 8
my_srand PROC
        mov     edx, DWORD PTR _tls_index
        mov     rax, QWORD PTR gs:88 ; 58h
        mov     r8d, OFFSET FLAT:rand_state
        mov     rax, QWORD PTR [rax+rdx*8]
        mov     DWORD PTR [r8+rax], ecx
        ret     0
my_srand ENDP

my_rand PROC
        mov     rax, QWORD PTR gs:88 ; 58h
        mov     ecx, DWORD PTR _tls_index
        mov     edx, OFFSET FLAT:rand_state
        mov     rcx, QWORD PTR [rax+rcx*8]
        imul    eax, DWORD PTR [rcx+rdx], 1664525 ; 0019660dH
        add     eax, 1013904223         ; 3c6ef35fH
        mov     DWORD PTR [rcx+rdx], eax
        and     eax, 32767              ; 00007fffH
        ret     0
my_rand ENDP

_TEXT   ENDS
```

Initialisation des données TLS

Imaginons maintenant que nous voulons nous prémunir des erreurs de programmation en initialisant systématiquement la variable rand_state avec une valeur constante (ligne 9) :

```
1 #include <stdint.h>
2 #include <windows.h>
3 #include <winnt.h>
4 
5 // from the Numerical Recipes book:
6 #define RNG_a 1664525
7 #define RNG_c 1013904223
8 
9 __declspec( thread ) uint32_t rand_state=1234;
10 
11 void my_srand (uint32_t init)
12 {
13         rand_state=init;
14 }
15 
16 int my_rand ()
17 {
18         rand_state=rand_state*RNG_a;
19         rand_state=rand_state+RNG_c;
20         return rand_state & 0x7fff;
21 }
22 
23 int main()
24 {
25         printf ("%d\n", my_rand());
26 };
```

Le code ne semble pas différent de celui que nous avons étudié. Pourtant dans IDA nous constatons:

```
.tls:00404000 ; Segment type: Pure data
.tls:00404000 ; Segment permissions: Read/Write
.tls:00404000 _tls            segment para public 'DATA' use32
.tls:00404000                 assume cs:_tls
.tls:00404000                 ;org 404000h
.tls:00404000 TlsStart        db    0         ; DATA XREF:
    .rdata:TlsDirectory
.tls:00404001                 db    0
.tls:00404002                 db    0
.tls:00404003                 db    0
.tls:00404004                 dd 1234
.tls:00404008 TlsEnd          db    0         ; DATA XREF: .rdata:TlsEnd_ptr
...
```

La valeur 1234 est bien présente. Chaque fois qu'un nouveau thread est créé, un nouvel espace TLS est alloué pour ses besoins et toutes les données - y compris 1234 - y sont copiées.

Considérons le scénario hypothétique suivant:

- Le thread A démarre. Un espace TLS est créé pour ses besoins et la valeur 1234 est copiée dans `rand_state`.
- La fonction `my_rand()` est invoquée plusieurs fois par le thread A.
  La valeur de la variable `rand_state` est maintenant différente de 1234.
- Le thread B démarre. Un espace TLS est créé pour ses besoins et la valeur 1234 est copiée dans `rand_state`. Dans le thread A, la valeur de `rand_state` demeure différente de 1234.

Fonctions de rappel TLS

Mais comment procédons-nous si les variables conservées dans l'environnement TLS doivent être initialisées avec des valeurs qui ne sont pas constantes ?

Imaginons le scénario suivant: Il se peut que le programmeur oublie d'invoquer la fonction `my_srand()` pour initialiser le PRNG. Malgré cela, le générateur doit être initialisé avec une valeur réellement aléatoire et non pas avec 1234. C'est précisément dans ce genre de cas que les fonctions de rappel TLS sont utilisées.

Le code ci-dessous n'est pas vraiment portable du fait du bricolage, mais vous devriez comprendre l'idée.

Nous définissons une fonction (`tls_callback()`) qui doit être invoquée avant le démarrage du processus et/ou d'un thread.

Cette fonction initialise le PRNG avec la valeur retournée par la fonction `GetTickCount()`.

```
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state;

void my_srand (uint32_t init)
{
        rand_state=init;
}

void NTAPI tls_callback(PVOID a, DWORD dwReason, PVOID b)
{
        my_srand (GetTickCount());
}

#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK p_thread_callback = tls_callback;
#pragma data_seg()

int my_rand ()
{
```

```
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
}

int main()
{
        // rand_state is already initialized at the moment (using
    GetTickCount())
        printf ("%d\n", my_rand());
};
```

Voyons cela dans IDA:

Listing 6.17 : avec optimisation MSVC 2013

```
.text:00401020 TlsCallback_0   proc near               ; DATA XREF:
    .rdata:TlsCallbacks
.text:00401020                 call    ds:GetTickCount
.text:00401026                 push    eax
.text:00401027                 call    my_srand
.text:0040102C                 pop     ecx
.text:0040102D                 retn    0Ch
.text:0040102D TlsCallback_0   endp

...

.rdata:004020C0 TlsCallbacks    dd offset TlsCallback_0 ; DATA XREF:
    .rdata:TlsCallbacks_ptr

...

.rdata:00402118 TlsDirectory    dd offset TlsStart
.rdata:0040211C TlsEnd_ptr      dd offset TlsEnd
.rdata:00402120 TlsIndex_ptr    dd offset TlsIndex
.rdata:00402124 TlsCallbacks_ptr dd offset TlsCallbacks
.rdata:00402128 TlsSizeOfZeroFill dd 0
.rdata:0040212C TlsCharacteristics dd 300000h
```

Les fonctions de rappel TLS sont parfois utilisées par les mécanismes de décompression d'exécutable afin d'en rendre le fonctionnement plus obscure.

Cette pratique peut en laisser certains dans le noir parce qu'ils auront omis de considérer qu'un fragment de code a pu s'exécuter avant l'OEP[5].

Linux

Voyons maintenant comment une variable globale conservée dans l'espace de stockage propre au thread est déclarée avec GCC:

```
__thread uint32_t rand_state=1234;
```

[5]Original Entry Point

Il ne s'agit pas du modificateur standard C/C++ modifier, mais bien d'un modificateur spécifique à GCC [6].

Le sélecteur GS: est utilisé lui aussi pour accéder au TLS, mais d'une manière un peu différente:

Listing 6.18 : avec optimisation GCC 4.8.1 x86

```
.text:08048460 my_srand        proc near
.text:08048460
.text:08048460 arg_0           = dword ptr  4
.text:08048460
.text:08048460                 mov     eax, [esp+arg_0]
.text:08048464                 mov     gs:0FFFFFFFCh, eax
.text:0804846A                 retn
.text:0804846A my_srand        endp

.text:08048470 my_rand         proc near
.text:08048470                 imul    eax, gs:0FFFFFFFCh, 19660Dh
.text:0804847B                 add     eax, 3C6EF35Fh
.text:08048480                 mov     gs:0FFFFFFFCh, eax
.text:08048486                 and     eax, 7FFFh
.text:0804848B                 retn
.text:0804848B my_rand         endp
```

Pour en savoir plus: [Ulrich Drepper, *ELF Handling For Thread-Local Storage*, (2013)][7].

6.3 Appels systèmes (syscall-s)

Comme nous le savons, tous les processus d'un OS sont divisés en deux catégories: ceux qui ont un accès complet au matériel («kernel space» espace noyau) et ceux qui ne l'ont pas («user space» espace utilisateur).

Le noyau de l'OS et, en général, les drivers sont dans la première catégorie.

Toutes les applications sont d'habitude dans la seconde catégorie.

Par exemple, le noyai Linux est dans le *kernel space*, mais la Glibc est dans le *user space*.

Cette séparation est cruciale pour la sécurité de l'OS : il est très important de ne pas donner la possibilité à n'importe quel processus de casser quelque chose dans un autre processus ou même dans le noyau de l'OS. D'un autre côté, un driver qui plante ou une erreur dans le noyau de l'OS se termine en général par un kernel panic ou un BSOD[8].

La protection en anneau dans les processeurs x86 permet de séparer l'ensemble dans quatre niveaux de protection (rings), mais tant dans Linux que Windows, seuls deux d'entre eux sont utilisés: ring0 («kernel space») et ring3 («user space»).

[6] https://gcc.gnu.org/onlinedocs/gcc-3.3/gcc/C99-Thread-Local-Edits.html
[7] Aussi disponible en http://www.akkadia.org/drepper/tls.pdf
[8] Blue Screen of Death

Les appels systèmes (syscall-s) sont un point où deux espaces sont connectés.

On peut dire que c'est la principale API fournie aux applications.

Comme dans Windows NT, la table des appels système se trouve dans la SSDT[9].

L'utilisation des appels système est très répandue parmi les auteurs de shellcode et de virus, car il est difficile de déterminer l'adresse des fonctions nécessaires dans les bibliothèques système, mais il est simple d'utiliser les appels système. Toutefois, il est nécessaire d'écrire plus de code à cause du niveau d'abstraction plus bas de l'API.

Il faut aussi noter que les numéros des appels systèmes peuvent être différent dans différentes versions d'un OS.

6.3.1 Linux

Sous Linux, un appel système s'effectue d'habitude par `int 0x80`. Le numéro de l'appel est passé dans le registre EAX, et tout autre paramètre —dans les autres registres.

Listing 6.19 : Un exemple simple de l'utilisation de deux appels système

```
section .text
global  _start

_start:
        mov     edx,len ; buffer len
        mov     ecx,msg ; buffer
        mov     ebx,1   ; file descriptor. 1 is for stdout
        mov     eax,4   ; syscall number. 4 is for sys_write
        int     0x80

        mov     eax,1   ; syscall number. 1 is for sys_exit
        int     0x80

section .data

msg     db  'Hello, world!',0xa
len     equ $ - msg
```

Compilation:

```
nasm -f elf32 1.s
ld 1.o
```

La liste complète des appels systèmes sous Linux: http://syscalls.kernelgrok.com/.

Pour intercepter les appels système et les suivre sous Linux, strace(7.2.3 on page 1032) peut être utilisé.

[9] System Service Dispatch Table

6.3.2 Windows

Ici, ils sont appelés via `int 0x2e` ou en utilisant l'instruction x86 spéciale SYSENTER.

La liste complète des appels systèmes sous Windows: http://j00ru.vexillium.org/ntapi/.

Pour aller plus loin:

«Windows Syscall Shellcode » par Piotr Bania: http://www.symantec.com/connect/articles/windows-syscall-shellcode.

6.4 Linux

6.4.1 Code indépendant de la position

Lorsque l'on analyse des bibliothèques partagées sous Linux (.so), on rencontre souvent ce genre de code:

Listing 6.20 : libc-2.17.so x86

```
.text:0012D5E3 __x86_get_pc_thunk_bx proc near   ; CODE XREF: sub_17350+3
.text:0012D5E3                                   ; sub_173CC+4 ...
.text:0012D5E3      mov     ebx, [esp+0]
.text:0012D5E6      retn
.text:0012D5E6 __x86_get_pc_thunk_bx endp

...

.text:000576C0 sub_576C0       proc near         ; CODE XREF: tmpfile+73

...

.text:000576C0      push    ebp
.text:000576C1      mov     ecx, large gs:0
.text:000576C8      push    edi
.text:000576C9      push    esi
.text:000576CA      push    ebx
.text:000576CB      call    __x86_get_pc_thunk_bx
.text:000576D0      add     ebx, 157930h
.text:000576D6      sub     esp, 9Ch

...

.text:000579F0      lea     eax, (a__gen_tempname - 1AF000h)[ebx] ;
    "__gen_tempname"
.text:000579F6      mov     [esp+0ACh+var_A0], eax
.text:000579FA      lea     eax, (a__SysdepsPosix - 1AF000h)[ebx] ;
    "../sysdeps/posix/tempname.c"
.text:00057A00      mov     [esp+0ACh+var_A8], eax
.text:00057A04      lea     eax, (aInvalidKindIn_ - 1AF000h)[ebx] ;
    "! \"invalid KIND in __gen_tempname\""
.text:00057A0A      mov     [esp+0ACh+var_A4], 14Ah
.text:00057A12      mov     [esp+0ACh+var_AC], eax
```

```
.text:00057A15      call    __assert_fail
```

Tous les pointeurs sur des chaînes sont corrigés avec une certaine constante et la valeur de EBX, qui est calculée au début de chaque fonction.

C'est ce que l'on appelle du code PIC, qui peut être exécuté à n'importe quelle adresse mémoire, c'est pourquoi il ne doit contenir aucune adresse absolue.

PIC était crucial dans les premiers ordinateurs, et l'est encore aujourd'hui dans les systèmes embarqués sans support de la mémoire virtuelle (où tous les processus se trouvent dans un seul bloc continu de mémoire).

C'est encore utilisé sur les systèmes *NIX pour les bibliothèques partagées, car elles sont utilisées par de nombreux processus mais ne sont chargées qu'une seule fois en mémoire. Mais tous ces processus peuvent mapper la même bibliothèque partagée à des adresses différentes, c'est pourquoi elles doivent fonctionner correctement sans aucune adresse absolue.

Faisons un petit essai:

```
#include <stdio.h>

int global_variable=123;

int f1(int var)
{
    int rt=global_variable+var;
    printf ("returning %d\n", rt);
    return rt;
};
```

Compilons le avec GCC 4.7.3 et examinons le fichier .so généré dans IDA :

```
gcc -fPIC -shared -O3 -o 1.so 1.c
```

Listing 6.21 : GCC 4.7.3

```
.text:00000440                 public __x86_get_pc_thunk_bx
.text:00000440 __x86_get_pc_thunk_bx proc near         ;
    CODE XREF: _init_proc+4
.text:00000440                                         ;
    deregister_tm_clones+4 ...
.text:00000440                 mov     ebx, [esp+0]
.text:00000443                 retn
.text:00000443 __x86_get_pc_thunk_bx endp

.text:00000570                 public f1
.text:00000570 f1              proc near
.text:00000570
.text:00000570 var_1C          = dword ptr -1Ch
.text:00000570 var_18          = dword ptr -18h
.text:00000570 var_14          = dword ptr -14h
.text:00000570 var_8           = dword ptr -8
.text:00000570 var_4           = dword ptr -4
```

```
.text:00000570 arg_0           = dword ptr  4
.text:00000570
.text:00000570                 sub     esp, 1Ch
.text:00000573                 mov     [esp+1Ch+var_8], ebx
.text:00000577                 call    __x86_get_pc_thunk_bx
.text:0000057C                 add     ebx, 1A84h
.text:00000582                 mov     [esp+1Ch+var_4], esi
.text:00000586                 mov     eax, ds:(global_variable_ptr - 2000h↙
    ↘ )[ebx]
.text:0000058C                 mov     esi, [eax]
.text:0000058E                 lea     eax, (aReturningD - 2000h)[ebx] ;
    "returning %d\n"
.text:00000594                 add     esi, [esp+1Ch+arg_0]
.text:00000598                 mov     [esp+1Ch+var_18], eax
.text:0000059C                 mov     [esp+1Ch+var_1C], 1
.text:000005A3                 mov     [esp+1Ch+var_14], esi
.text:000005A7                 call    ___printf_chk
.text:000005AC                 mov     eax, esi
.text:000005AE                 mov     ebx, [esp+1Ch+var_8]
.text:000005B2                 mov     esi, [esp+1Ch+var_4]
.text:000005B6                 add     esp, 1Ch
.text:000005B9                 retn
.text:000005B9 f1              endp
```

C'est ça: les pointeurs sur *«returning %d\n»* et *global_variable* sont corrigés à chaque exécution de la fonction.

La fonction `__x86_get_pc_thunk_bx()` renvoie dans EBX l'adresse de l'instruction après son appel (`0x57C` ici).

C'est un moyen simple d'obtenir la valeur du compteur de programme (EIP) à un endroit quelconque. La constante `0x1A84` est relative à la différence entre le début de cette fonction et ce que l'on appelle *Global Offset Table Procedure Linkage Table* (GOT PLT), la section juste après la *Global Offset Table* (GOT), où se trouve le pointeur sur *global_variable*. IDA montre ces offsets sous leur forme calculée pour rendre les choses plus facile à comprendre, mais en fait, le code est:

```
.text:00000577                 call    __x86_get_pc_thunk_bx
.text:0000057C                 add     ebx, 1A84h
.text:00000582                 mov     [esp+1Ch+var_4], esi
.text:00000586                 mov     eax, [ebx-0Ch]
.text:0000058C                 mov     esi, [eax]
.text:0000058E                 lea     eax, [ebx-1A30h]
```

Ici, EBX pointe sur la section `GOT` PLT et pour calculer le pointeur sur *global_variable* (qui est stocké dans la `GOT`), il faut soustraire `0xC`.

Pour calculer la valeur du pointeur sur la chaîne *«returning %d\n»*, il faut soustraire `0x1A30`.

A propos, c'est la raison pour laquelle le jeu d'instructions AMD64 ajoute le support d'adressage relatif de RIP[10] — pour simplifier le code PIC.

[10]compteur de programme sur AMD64

Compilons le même code C en utilisant la même version de GCC, mais pour x64.

IDA simplifierait le code en supprimant les détails de l'adressage relatif à RIP, donc utilisons *objdump* à la place d'IDA pour tout voir:

```
0000000000000720 <f1>:
 720:   48 8b 05 b9 08 20 00    mov    rax,QWORD PTR [rip+0x2008b9]       ↙
    ↳ ;
    200fe0 <_DYNAMIC+0x1d0>
 727:   53                      push   rbx
 728:   89 fb                   mov    ebx,edi
 72a:   48 8d 35 20 00 00 00    lea    rsi,[rip+0x20]        ;
    751 <_fini+0x9>
 731:   bf 01 00 00 00          mov    edi,0x1
 736:   03 18                   add    ebx,DWORD PTR [rax]
 738:   31 c0                   xor    eax,eax
 73a:   89 da                   mov    edx,ebx
 73c:   e8 df fe ff ff          call   620 <__printf_chk@plt>
 741:   89 d8                   mov    eax,ebx
 743:   5b                      pop    rbx
 744:   c3                      ret
```

`0x2008b9` est la différence entre l'adresse de l'instruction en `0x720` et *global_variable*, et `0x20` est la différence entre l'adresse de l'instruction en `0x72A` et la chaîne «*returning %d\n*».

Comme on le voit, le fait d'avoir à recalculer fréquemment les adresses rend l'exécution plus lente (cela dit, ça c'est amélioré en x64).

Donc, il est probablement mieux de lier statiquement si vous vous préoccupez des performances [voir: Agner Fog, *Optimizing software in C++* (2015)].

Windows

Le mécanisme PIC n'est pas utilisé dans les DLLs de Windows. Si le chargeur de Windows doit charger une DLL à une autre adresse, il «patche» la DLL en mémoire (aux places *FIXUP*) afin de corriger toutes les adresses.

Cela implique que plusieurs processus Windows ne peuvent pas partager une DLL déjà chargée à une adresse différente dans des blocs mémoire de différents processus — puisque chaque instance qui est chargée en mémoire est *fixé* pour fonctionner uniquement à ces adresses...

6.4.2 Hack *LD_PRELOAD* sur Linux

Cela permet de charger nos propres bibliothèques dynamiques avant les autres, même celle du système, comme libc.so.6.

Cela permet de «substituer» nos propres fonctions, avant celles originales des bibliothèques du système. Par exemple, il est facile d'intercepter tous les appels à time(), read(), write(), etc.

Regardons si l'on peut tromper l'utilitaire *uptime*. Comme chacun le sait, il indique

depuis combien de temps l'ordinateur est démarré. Avec l'aide de strace(7.2.3 on page 1032), on voit que ces informations proviennent du fichier `/proc/uptime` :

```
$ strace uptime
...
open("/proc/uptime", O_RDONLY)          = 3
lseek(3, 0, SEEK_SET)                   = 0
read(3, "416166.86 414629.38\n", 2047)  = 20
...
```

Ce n'est pas un fichier réel sur le disque, il est virtuel et généré au vol par le noyau Linux. Il y a seulement deux nombres:

```
$ cat /proc/uptime
416690.91 415152.03
```

Ce que l'on peut apprendre depuis Wikipédia [11] :

> Le premier est le nombre total de seconde depuis lequel le système est démarré. Le second est la part de ce temps pendant lequel la machine était inoccupée, en secondes.

Essayons d'écrire notre propre bibliothèque dynamique, avec les fonctions open(), read(), close() fonctionnant comme nous en avons besoin.

Tout d'abord, notre fonction open() va comparer le nom du fichier à ouvrir avec celui que l'on veut modifier, et si c'est le cas, sauver le descripteur du fichier ouvert.

Ensuite, si read() est appelé avec ce descripteur de fichier, nous allons substituer la sortie, et dans les autres cas, nous appellerons la fonction read() originale de libc.so.6. Et enfin close() fermera le fichier que nous suivons.

Nous allons utiliser les fonctions dlopen() et dlsym() pour déterminer les adresses des fonctions originales dans libc.so.6.

Nous en avons besoin pour appeler les fonctions «réelles».

D'un autre côté, si nous interceptons strcmp() et surveillons chaque comparaison de chaînes dans le programme, nous pouvons alors implémenter notre propre fonction strcmp(), et ne pas utiliser la fonction originale du tout.

[12].

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>
```

[11] Wikipédia
[12] Par exemple, voilà comment implémenter une interception simple de strcmp() dans cet article [13] écrit par Yong Huang

```
void *libc_handle = NULL;
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void*, size_t) = NULL;

bool inited = false;

_Noreturn void die (const char * fmt, ...)
{
        va_list va;
        va_start (va, fmt);

        vprintf (fmt, va);
        exit(0);
};

static void find_original_functions ()
{
        if (inited)
                return;

        libc_handle = dlopen ("libc.so.6", RTLD_LAZY);
        if (libc_handle==NULL)
                die ("can't open libc.so.6\n");

        open_ptr = dlsym (libc_handle, "open");
        if (open_ptr==NULL)
                die ("can't find open()\n");

        close_ptr = dlsym (libc_handle, "close");
        if (close_ptr==NULL)
                die ("can't find close()\n");

        read_ptr = dlsym (libc_handle, "read");
        if (read_ptr==NULL)
                die ("can't find read()\n");

        inited = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags)
{
        find_original_functions();

        int fd=(*open_ptr)(pathname, flags);
        if (strcmp(pathname, "/proc/uptime")==0)
                opened_fd=fd; // that's our file! record its file descriptor
        else
                opened_fd=0;
        return fd;
```

```
};

int close(int fd)
{
        find_original_functions();

        if (fd==opened_fd)
                opened_fd=0; // the file is not opened anymore
        return (*close_ptr)(fd);
};

ssize_t read(int fd, void *buf, size_t count)
{
        find_original_functions();

        if (opened_fd!=0 && fd==opened_fd)
        {
                // that's our file!
                return snprintf (buf, count, "%d %d", 0x7fffffff, 0↙
    ↘ x7fffffff)+1;
        };
        // not our file, go to real read() function
        return (*read_ptr)(fd, buf, count);
};
```

(Code source)

Compilons le comme une bibliothèque dynamique standard:

```
gcc -fpic -shared -Wall -o fool_uptime.so fool_uptime.c -ldl
```

Lançons *uptime* en chargeant notre bibliothèque avant les autres:

```
LD_PRELOAD=`pwd`/fool_uptime.so uptime
```

Et nous voyons:

```
 01:23:02 up 24855 days,  3:14,  3 users,  load average: 0.00, 0.01, 0.05
```

Si la variable d'environnement *LD_PRELOAD*

pointe toujours sur le nom et le chemin de notre bibliothèque, elle sera chargée pour tous les programmes démarrés.

Plus d'exemples:

- Interception simple de strcmp() (Yong Huang) https://yurichev.com/mirrors/LD_PRELOAD/Yong%20Huang%20LD_PRELOAD.txt
- Kevin Pulo—Jouons avec LD_PRELOAD. De nombreux exemples et idées. yurichev.com
- Interception des fonctions de fichier pour compresser/décompresser les fichiers au vol (zlibc) ftp://metalab.unc.edu/pub/Linux/libs/compression

6.5 Windows NT

6.5.1 CRT (win32)

L'exécution du programme débute donc avec la fonction `main()` ? Non, pas du tout.

Ouvrez un exécutable dans IDA ou HIEW, et vous constaterez que l'OEP pointe sur un bloc de code qui se situe ailleurs.

Ce code prépare l'environnement avant de passer le contrôle à notre propre code. Ce fragment de code initial est dénommé CRT (pour C RunTime).

La fonction `main()` accepte en arguments un tableau des paramètres passés en ligne de commande, et un autre contenant les variables d'environnement. En réalité, ce qui est passé au programme est une simple chaîne de caractères. Le code du CRT repère les espaces dans la chaîne et découpe celle-ci en plusieurs morceaux qui constitueront le tableau des paramètres. Le CRT est également responsable de la préparation du tableau des variables d'environnement envp.

En ce qui concerne les applications win32 dotées d'un interface graphique (GUI[14]). La fonction principale est nommée `WinMain` et non pas `main()`. Elle possède aussi ses propres arguments:

```
int CALLBACK WinMain(
  _In_  HINSTANCE hInstance,
  _In_  HINSTANCE hPrevInstance,
  _In_  LPSTR lpCmdLine,
  _In_  int nCmdShow
);
```

C'est toujours le CRT qui est responsable de leurs préparation.

La valeur retournée par la fonction `main()` est également ce qui constitue le code retour du programme.

Le CRT peut utiliser cette valeur lors de l'appel de la fonction `ExitProcess()` qui prend le code retour du programme en paramètre.

En règle générale, le code du CRT est propre à chaque compilateur.

Voici celui du MSVC 2008.

```
___tmainCRTStartup proc near

var_24 = dword ptr -24h
var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
ms_exc = CPPEH_RECORD ptr -18h

       push    14h
       push    offset stru_4092D0
       call    __SEH_prolog4
```

[14] Graphical User Interface

```
        mov     eax, 5A4Dh
        cmp     ds:400000h, ax
        jnz     short loc_401096
        mov     eax, ds:40003Ch
        cmp     dword ptr [eax+400000h], 4550h
        jnz     short loc_401096
        mov     ecx, 10Bh
        cmp     [eax+400018h], cx
        jnz     short loc_401096
        cmp     dword ptr [eax+400074h], 0Eh
        jbe     short loc_401096
        xor     ecx, ecx
        cmp     [eax+4000E8h], ecx
        setnz   cl
        mov     [ebp+var_1C], ecx
        jmp     short loc_40109A


loc_401096: ; CODE XREF: ___tmainCRTStartup+18
            ; ___tmainCRTStartup+29 ...
        and     [ebp+var_1C], 0

loc_40109A: ; CODE XREF: ___tmainCRTStartup+50
        push    1
        call    __heap_init
        pop     ecx
        test    eax, eax
        jnz     short loc_4010AE
        push    1Ch
        call    _fast_error_exit
        pop     ecx

loc_4010AE: ; CODE XREF: ___tmainCRTStartup+60
        call    __mtinit
        test    eax, eax
        jnz     short loc_4010BF
        push    10h
        call    _fast_error_exit
        pop     ecx

loc_4010BF: ; CODE XREF: ___tmainCRTStartup+71
        call    sub_401F2B
        and     [ebp+ms_exc.disabled], 0
        call    __ioinit
        test    eax, eax
        jge     short loc_4010D9
        push    1Bh
        call    __amsg_exit
        pop     ecx

loc_4010D9: ; CODE XREF: ___tmainCRTStartup+8B
        call    ds:GetCommandLineA
        mov     dword_40B7F8, eax
```

```
        call    ___crtGetEnvironmentStringsA
        mov     dword_40AC60, eax
        call    __setargv
        test    eax, eax
        jge     short loc_4010FF
        push    8
        call    __amsg_exit
        pop     ecx

loc_4010FF: ; CODE XREF: ___tmainCRTStartup+B1
        call    __setenvp
        test    eax, eax
        jge     short loc_401110
        push    9
        call    __amsg_exit
        pop     ecx

loc_401110: ; CODE XREF: ___tmainCRTStartup+C2
        push    1
        call    __cinit
        pop     ecx
        test    eax, eax
        jz      short loc_401123
        push    eax
        call    __amsg_exit
        pop     ecx

loc_401123: ; CODE XREF: ___tmainCRTStartup+D6
        mov     eax, envp
        mov     dword_40AC80, eax
        push    eax             ; envp
        push    argv            ; argv
        push    argc            ; argc
        call    _main
        add     esp, 0Ch
        mov     [ebp+var_20], eax
        cmp     [ebp+var_1C], 0
        jnz     short $LN28
        push    eax             ; uExitCode
        call    $LN32

$LN28:      ; CODE XREF: ___tmainCRTStartup+105
        call    __cexit
        jmp     short loc_401186


$LN27:      ; DATA XREF: .rdata:stru_4092D0
        mov     eax, [ebp+ms_exc.exc_ptr] ; Exception filter 0 for function 401044
        mov     ecx, [eax]
        mov     ecx, [ecx]
        mov     [ebp+var_24], ecx
        push    eax
        push    ecx
```

```
117         call    __XcptFilter
118         pop     ecx
119         pop     ecx
120
121 $LN24:
122         retn
123
124
125 $LN14:       ; DATA XREF: .rdata:stru_4092D0
126         mov     esp, [ebp+ms_exc.old_esp] ; Exception handler 0 for function
    401044
127         mov     eax, [ebp+var_24]
128         mov     [ebp+var_20], eax
129         cmp     [ebp+var_1C], 0
130         jnz     short $LN29
131         push    eax              ; int
132         call    __exit
133
134
135 $LN29:       ; CODE XREF: ___tmainCRTStartup+135
136         call    __c_exit
137
138 loc_401186: ; CODE XREF: ___tmainCRTStartup+112
139         mov     [ebp+ms_exc.disabled], 0FFFFFFFEh
140         mov     eax, [ebp+var_20]
141         call    __SEH_epilog4
142         retn
```

Nous y trouvons des appels à `GetCommandLineA()` (line 62), puis `setargv()` (line 66) et `setenvp()` (line 74), qui semblent être utilisés pour initialiser les variables globales `argc`, `argv`, `envp`.

Enfin, `main()` est invoqué avec ces arguments (line 97).

Nous observons également des appels à des fonctions au nom évocateur telles que `heap_init()` (line 35), `ioinit()` (line 54).

Le tas heap est lui aussi initialisé par le CRT. Si vous tentez d'utiliser la fonction `malloc()` dans un programme ou le CRT n'a pas été ajouté, le programme va s'achever de manière anormale avec l'erreur suivante:

```
runtime error R6030
- CRT not initialized
```

L'initialisation des objets globaux en C++ est également de la responsabilité du CRT avant qu'il ne passe la main à `main()` : 3.21.4 on page 729.

La valeur retournée par `main()` est passée soit à la fonction `cexit()`, soit à `$LN32` qui à son tour invoque `doexit()`.

Mais pourrait-on se passe du CRT ? Oui, si vous savez ce que vous faites.

L'éditeur de lien MSVC accepte une option `/ENTRY` qui permet de définir le point d'entrée du programme.

```
#include <windows.h>

int main()
{
        MessageBox (NULL, "hello, world", "caption", MB_OK);
};
```

Compilons ce programme avec MSVC 2008.

```
cl no_crt.c user32.lib /link /entry:main
```

Nous obtenons un programme exécutable .exe de 2560 octets qui contient en tout et pour tout l'en-tête PE, les instructions pour invoquer MessageBox et deux chaînes de caractères dans le segment de données: le nom de la fonction MessageBox et celui de sa DLL d'origine user32.dll.

Cela fonctionne, mais vous ne pouvez pas déclarer WinMain et ses 4 arguments à la place de main().

Pour être précis, vous pourriez, mais les arguments ne seraient pas préparés au moment de l'exécution du programme.

Cela étant, il est possible de réduire encore la taille de l'exécutable en utilisant une valeur pour l'alignement des sections du fichier au format PE une valeur inférieur à celle par défaut de 4096 octets.

```
cl no_crt.c user32.lib /link /entry:main /align:16
```

L'éditeur de lien prévient cependant:

```
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image may not run
```

Nous pouvons ainsi obtenir un exécutable de 720 octets. Son exécution est possible sur Windows 7 x86, mais pas en environnement x64 (un message d'erreur apparaîtra alors si vous tentez de l'exécuter).

Avec des efforts accrus il est possible de réduire encore la taille, mais avec des problèmes de compatibilité comme vous le voyez.

6.5.2 Win32 PE

PE est un format de fichier exécutable utilisé sur Windows. La différence entre .exe, .dll et .sys est que les .exe et .sys n'ont en général pas d'exports, uniquement des imports.

Une DLL[15], tout comme n'importe quel fichier PE, possède un point d'entrée (OEP) (la fonction DllMain() se trouve là) mais cette fonction ne fait généralement rien. .sys est généralement un pilote de périphérique. Pour les pilotes, Windows exige que la somme de contrôle soit présente dans le fichier PE et qu'elle soit correcte [16].

[15] Dynamic-Link Library
[16] Par exemple, Hiew(7.1 on page 1029) peut la calculer

A partir de Windows Vista, un fichier de pilote doit aussi être signé avec une signature numérique. Le chargement échouera sinon.

Chaque fichier PE commence avec un petit programme DOS qui affiche un message comme «Ce programme ne peut pas être lancé en mode DOS. »—si vous lancez sous DOS ou Windows 3.1 (OS-es qui ne supportent pas le format PE), ce message sera affiché.

Terminologie

- Module—un fichier séparé, .exe ou .dll.
- Process—un programme chargé dans la mémoire et fonctionnant actuellement. Consiste en général d'un fichier .exe et d'un paquet de fichiers .dll.
- Process memory—la mémoire utilisée par un processus. Chaque processus a la sienne. Il y a en général des modules chargés, la mémoire pour la pile, tas(s), etc.
- VA[17]—une adresse qui est utilisée pendant le déroulement du programme.
- Base address (du module)—l'adresse dans la mémoire du processus à laquelle le module est chargé. Le chargeur de l'OS peut la changer, si l'adresse de base est occupée par un module déjà chargé.
- RVA[18]—l'adresse VA-moins l'adresse de base.

  De nombreuses adresses dans les fichiers PE utilisent l'adresse RVA.
- IAT[19]—un tableau d'adresses des symboles importés [20].
  Parfois, le répertoire de données `IMAGE_DIRECTORY_ENTRY_IAT` pointe sur l'IAT. Il est utile de noter qu'IDA (à partir de 6.1) peut allouer une pseudo-section nommée `.idata` pour l'IAT, même si l'IAT fait partie d'une autre section!
- INT[21]—une tableau de noms de symboles qui doivent être importés[22].

Adresse de base

Le problème est que plusieurs auteurs de module peuvent préparer des fichiers DLL pour que d'autres les utilisent et qu'il n'est pas possible de s'accorder pour assigner une adresse à ces modules.

C'est pourquoi si deux DLLs nécessaires pour un processus ont la même adresse de base, une des deux sera chargée à cette adresse de base, et l'autre—à un autre espace libre de la mémoire du processus et chaque adresse virtuelle de la seconde DLL sera corrigée.

[17]Virtual Address
[18]Relative Virtual Address
[19]Import Address Table
[20]Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]
[21]Import Name Table
[22]Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]

Avec MSVC l'éditeur de lien génère souvent les fichiers .exe avec une adresse de base en `0x400000`[23], et avec une section de code débutant en `0x401000`. Cela signifie que la RVA du début de la section de code est `0x1000`.

Les DLLs sont souvent générées par l'éditeur de lien de MSVC avec une adresse de base de `0x10000000`[24].

Il y a une autre raison de charger les modules a des adresses de base variées, aléatoires dans ce cas. C'est l'ASLR[25].

Un shellcode essayant d'être exécuté sur une système compromis doit appeler des fonctions systèmes, par conséquent, connaître leurs adresses.

Dans les anciens OS (dans la série Windows NT : avant Windows Vista), les DLLs système (comme kernel32.dll, user32.dll) étaient toujours chargées à une adresse connue, et si nous nous rappelons que leur version changeait rarement, l'adresse des fonctions était fixe et le shellcode pouvait les appeler directement.

Pour éviter ceci, la méthode ASLR charge votre programme et tous les modules dont il a besoin à des adresses de base aléatoires, différentes à chaque fois.

Le support de l'ASLR est indiqué dans un fichier PE en mettant le flag `IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE` [voir Mark Russinovich, *Microsoft Windows Internals*].

Sous-système

Il a aussi un champ *sous-système*, d'habitude c'est:

- natif[26] (.sys-driver),
- console (application en console) ou
- GUI (non-console).

Version d'OS

Un fichier PE indique aussi la version de Windows minimale requise pour pouvoir être chargé.

La table des numéros de version stockés dans un fichier PE et les codes de Windows correspondants est ici[27].

Par exemple, MSVC 2005 compile les fichiers .exe pour tourner sur Windows NT4 (version 4.00), mais MSVC 2008 ne le fait pas (les fichiers générés ont une version de 5.00, il faut au moins Windows 2000 pour les utiliser).

[23] L'origine de ce choix d'adresse est décrit ici: MSDN
[24] Cela peut être changé avec l'option /BASE de l'éditeur de liens
[25] Wikipédia
[26] Signifiant que le module utilise l'API Native au lieu de Win32
[27] Wikipédia

MSVC 2012 génère des fichiers .exe avec la version 6.00 par défaut, visant au moins Windows Vista. Toutefois, en changeant l'option du compilateur[28], il est possible de le forcer à compiler pour Windows XP.

Sections

Il semble que la division en section soit présente dans tous les formats de fichier exécutable.

C'est divisé afin de séparer le code des données, et les données—des données constantes.

- Si l'un des flags *IMAGE_SCN_CNT_CODE* ou *IMAGE_SCN_MEM_EXECUTE* est mis dans la section de code—c'est de code exécutable.
- Dans la section de données—les flags *IMAGE_SCN_CNT_INITIALIZED_DATA*, *IMAGE_SCN_MEM_READ* et *IMAGE_SCN_MEM_WRITE*.
- Dans une section vide avec des données non initialisées—*IMAGE_SCN_CNT_UNINITIALIZED_DATA*, *IMAGE_SCN_MEM_READ* et *IMAGE_SCN_MEM_WRITE*.
- Dans une section de données constantes (une qui est protégée contre l'écriture), les flags *IMAGE_SCN_CNT_INITIALIZED_DATA* et *IMAGE_SCN_MEM_READ* peuvent être mis, mais pas *IMAGE_SCN_MEM_WRITE*. Un processus plantera s'il essaye d'écrire dans cette section.

Chaque section dans un fichier PE peut avoir un nom, toutefois, ce n'est pas très important. (peut-être que `.rdata` signifie *read-only-data*). Souvent (mais pas toujours) la section de code est appelée `.text`, la section de données—`.data`, la section de données constante — `.rdata` *(readable data)*. D'autres noms de section courants sont:

- `.idata`—section d'imports. IDA peut créer une pseudo-section appelée ainsi: 6.5.2 on page 986.
- `.edata`—section d'exports section (rare)
- `.pdata`—section contenant toutes les informations sur les exceptions dans Windows NT pout MIPS, IA64 et x64: 6.5.3 on page 1021
- `.reloc`—section de relocalisation
- `.bss`—données non initialisées (BSS)
- `.tls`—stockage local d'un thread (TLS)
- `.rsrc`—ressources
- `.CRT`—peut être présente dans les fichiers binaire compilés avec des anciennes versions de MSVC

Les packeurs/chiffreurs rendent souvent les noms de sections inintelligibles ou les remplacent avec des noms qui leurs sont propres.

[28] MSDN

MSVC permet de déclarer des données dans une section de n'importe quel nom [29].

Certains compilateurs et éditeurs de liens peuvent ajouter une section avec des symboles et d'autres informations de débogage (MinGW par exemple). Toutefois ce n'est pas comme cela dans les dernières versions de MSVC (des fichiers PDB séparés sont utilisés dans ce but).

Voici comment une section PE est décrite dans le fichier:

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

[30]

Un mot à propos de le terminologie: *PointerToRawData* est appelé «Offset» dans Hiew et *VirtualAddress* est appelé «RVA» ici.

.rdata — section de données en lecture seule (read-only)

Les chaînes sont généralement situées ici (car elles ont le type `const char*`), ainsi que d'autres variables marquées comme *const*, les noms des fonctions importées.

Voir aussi: 3.3 on page 596.

Relocs (relocalisation)

AKA FIXUP-s (au moins dans Hiew).

Elles sont aussi présentes dans presque tous les formats de fichiers exécutables [31]. Les exceptions sont les bibliothèques dynamiques partagées compilées avec PIC, ou tout autre code PIC.

À quoi servent-elles?

Évidemment, les modules peuvent être chargés à différentes adresse de base, mais comment traiter les variables globales, par exemple? Elles doivent être accédées

[29] MSDN
[30] MSDN
[31] Même dans les fichiers .exe pour MS-DOS

par adresse. Une solution est le code indépendant de la position (6.4.1 on page 974). Mais ce n'est pas toujours pratique.

C'est pourquoi une table des relocalisations est présente. Elle contient les adresses des points qui doivent être corrigés en cas de chargement à une adresse de base différente.

Par exemple, il y a une variable globale à l'adresse 0x410000 et voici comment elle est accédée:

```
A1 00 00 41 00         mov         eax,[000410000]
```

L'adresse de base du module est 0x400000, le RVA de la variable globale est 0x10000.

Si le module est chargé à l'adresse de base 0x500000, l'adresse réelle de la variable globale doit être 0x510000.

Comme nous pouvons le voir, l'adresse de la variable est encodée dans l'instruction MOV, après l'octet 0xA1.

C'est pourquoi l'adresse des 4 octets aprés 0xA1 est écrite dans la table de relocalisations.

Si le module est chargé à une adresse de base différente, le chargeur de l'OS parcourt toutes les adresses dans la table, trouve chaque mot de 32-bit vers lequel elles pointent, soustrait l'adresse de base d'origine (nous obtenons le RVA ici), et ajoute la nouvelle adresse de base.

Si un module est chargé à son adresse de base originale, rien ne se passe.

Toutes les variables globales sont traitées ainsi.

Relocs peut avoir différent types, toutefois, dans Windows pour processeurs x86, le type est généralement
IMAGE_REL_BASED_HIGHLOW.

Á propos, les relocs sont plus foncés dans Hiew, par exemple: fig.1.21. (vous devez éviter ces octets lors d'un patch.)

OllyDbg souligne les endroits en mémoire où sont appliqués les relocs, par exemple: fig.1.52.

Exports et imports

Comme nous le savons, tout programme exécutable doit utiliser les services de l'OS et autres bibliothèques d'une manière ou d'une autre.

On peut dire que les fonctions d'un module (en général DLL) doivent être reliées aux points de leur appel dans les autres modules (fichier .exe ou autre DLL).

Pour cela, chaque DLL a une table d'«exports», qui consiste en les fonctions plus leur adresse dans le module.

Et chaque fichier .exe ou DLL possède des «imports», une table des fonctions requises pour l'exécution incluant une liste des noms de DLL.

Après le chargement du fichier .exe principal, le chargeur de l'OS traite la table des imports: il charge les fichiers DLL additionnels, trouve les noms des fonctions parmi les exports des DLL et écrit leur adresse dans le module IAT de l'.exe principal.

Comme on le voit, pendant le chargement, le chargeur doit comparer de nombreux noms de fonctions, mais la comparaison des chaînes n'est pas une procédure très rapide, donc il y a un support pour «ordinaux» ou «hints», qui sont les numéros de fonctions stockés dans une table au lieu de leurs noms.

C'est ainsi qu'elles peuvent être localisées plus rapidement lors du chargement d'une DLL. Les ordinaux sont toujours présents dans la table d'«export».

Par exemple, un programme utilisant la bibliothèque MFC[32] charge en général mfc*.dll par ordinaux, et dans un programme n'ayant aucun nom de fonction MFC dans INT.

Lors du chargement d'un tel programme dans IDA, il va demander le chemin vers les fichiers mfc*.dll, afin de déterminer le nom des fonctions.

Si vous ne donnez pas le chemin vers ces DLLs à IDA, il y aura *mfc80_123* à la place des noms de fonctions.

Section d'imports

Souvent, une section séparée est allouée pour la table d'imports et tout ce qui y est relatif (avec un nom comme `.idata`), toutefois, ce n'est pas une règle stricte.

Les imports sont un sujet prêtant à confusion à cause de la terminologie confuse. Essayons de regrouper toutes les informations au même endroit.

[32] Microsoft Foundation Classes

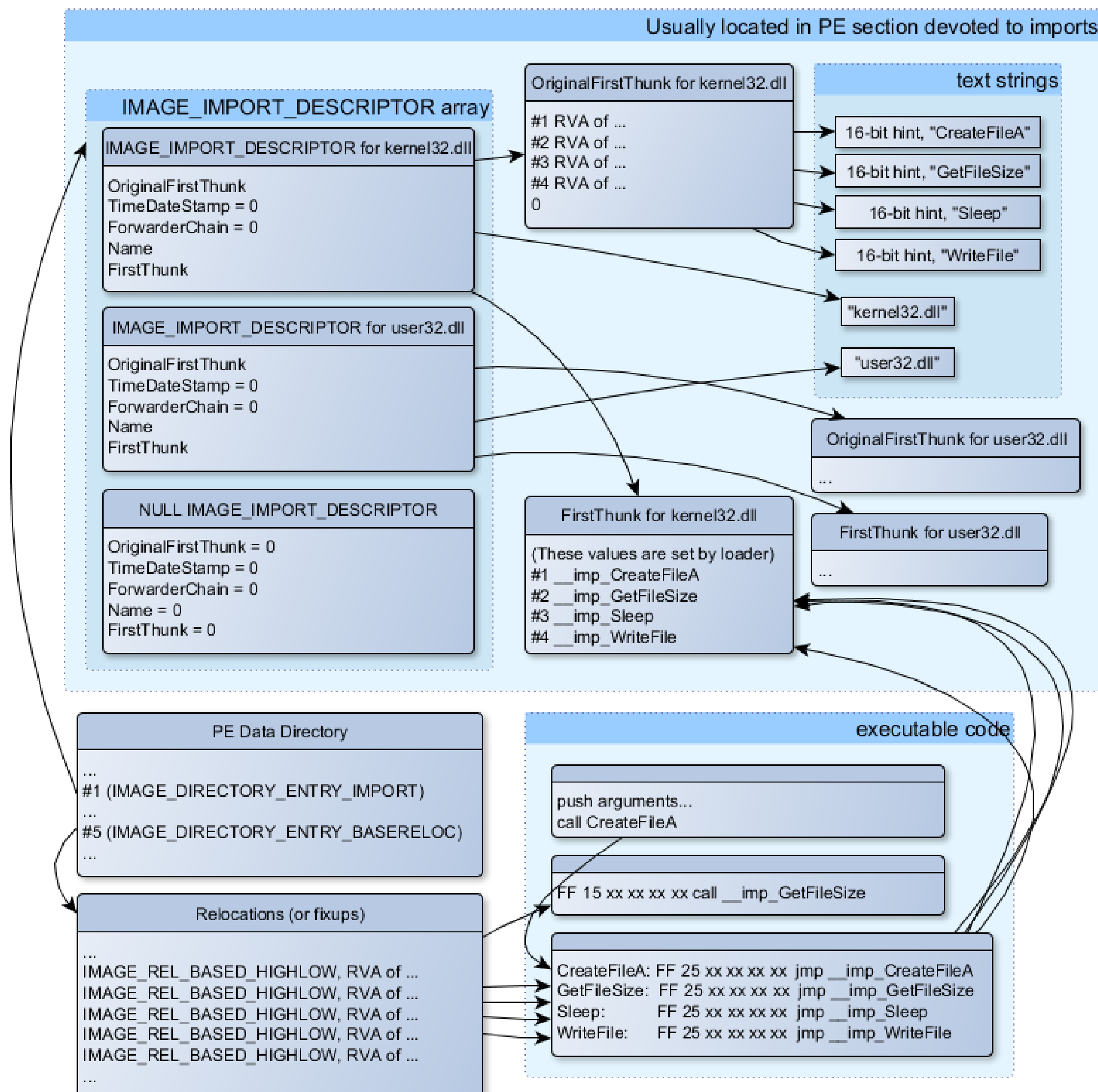


Fig. 6.1: Un schéma qui regroupe toutes les structures concernant les imports d'un fichier PE

La structure principale est le tableau *IMAGE_IMPORT_DESCRIPTOR*, qui comprend un élément pour chaque DLL importée.

Chaque élément contient l'adresse RVA de la chaîne de texte (nom de la DLL) (*Name*).

OriginalFirstThunk est l'adresse RVA de la table INT. C'est un tableau d'adresses RVA, qui pointe chacune sur une chaîne de texte avec le nom d'une fonction. Chaque chaîne est préfixée par un entier 16-bit («hint»)—«ordinal» de la fonction.

Pendant le chargement, s'il est possible de trouver une fonction par ordinal, la com-

paraison de chaînes n'a pas lieu. Le tableau est terminé par zéro.

Il y a aussi un pointeur sur la table IAT appelé *FirstThunk*, qui est juste l'adresse RVA de l'endroit où le chargeur écrit l'adresse résolue de la fonction.

Les endroits où le chargeur écrit les adresses sont marqués par IDA comme ceci: *__imp_CreateFileA*, etc.

Il y a au moins deux façons d'utiliser les adresses écrites par le chargeur.

- Le code aura des instructions comme *call __imp_CreateFileA*, et puisque le champ avec l'adresse de la fonction importée est en un certain sens une variable globale, l'adresse de l'instruction *call* (plus 1 ou 2) doit être ajoutée à la table de relocs, pour le cas où le module est chargé à une adresse de base différente.

  Mais, évidemment, ceci augmente significativement la table de relocs.

  Car il peut y avoir un grand nombre d'appels aux fonctions importées dans le module.

  En outre, une grosse table de relocs ralenti le processus de chargement des modules.

- Pour chaque fonction importée, il y a seulement un saut alloué, en utilisant l'instruction JMP ainsi qu'un reloc sur lui. De tel point sont aussi appelés «thunks».

  Tous les appels aux fonction importées sont juste des instructions CALL au «thunk» correspondant. Dans ce cas, des relocs supplémentaires ne sont pas nécessaire car cex CALL-s ont des adresses relatives et ne doivent pas être corrigés.

Ces deux méthodes peuvent être combinées.

Il est possible que l'éditeur de liens créé des «thunk»s individuels si il n'y a pas trop d'appels à la fonction, mais ce n'est pas fait par défaut

Á propos, le tableau d'adresses de fonction sur lequel pointe FirstThunk ne doit pas nécessairement se trouver dans la section IAT. Par exemple, j'ai écrit une fois l'utilitaire PE_add_import[33] pour ajouter des imports à un fichier .exe existant.

Quelques temps plus tôt, dans une version précédente de cet utilitaire, à la place de la fonction que vous vouliez substituer par un appel à une autre DLL, mon utilitaire écrivait le code suivant:

```
MOV EAX, [yourdll.dll!function]
JMP EAX
```

FirstThunk pointe sur la première instruction. En d'autres mots, en chargeant yourdll.dll, le chargeur écrit l'adresse de la fonction *function* juste après le code.

Il est à noter qu'une section de code est en général protégée contre l'écriture, donc mon utilitaire ajoute le flag

[33] yurichev.com

IMAGE_SCN_MEM_WRITE pour la section de code. Autrement, le programme planterait pendant le chargement avec le code d'erreur 5 (access denied).

On pourrait demander: pourrais-je fournir à un programme un ensemble de fichiers DLL qui n'est pas supposé changer (incluant les adresses des fonctions DLL), est-il possible d'accélérer le processus de chargement?

Oui, il est possible d'écrire les adresses des fonctions qui doivent être importées en avance dans le tableau FirstThunk.
Le champ *Timestamp* est présent dans la structure *IMAGE_IMPORT_DESCRIPTOR*.

Si une valeur est présente ici, alors le loader compare cette valeur avec la date et l'heure du fichier DLL.

Si les valeurs sont égales, alors le loader ne fait rien, et le processus de chargement peut être plus rapide. Ceci est appelé «old-style binding» [34].

L'utilitaire BIND.EXE dans le SDK est fait pour ça. Pour accélérer le chargement de votre programme, Matt Pietrek dans Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)][35], suggère de faire le binding juste après l'installation de votre programme sur l'ordinateur de l'utilisateur final.

Les packeurs/chiffreurs de fichier PE peuvent également compresser/chiffrer la table des imports.

Dans ce cas, le loader de Windows, bien sûr, ne va pas charger les DLLs nécessaires.

Par conséquent, le packeur/chiffreur fait cela lui même, avec l'aide des fonctions *LoadLibrary()* et *GetProcAddress()*.

C'est pourquoi ces deux fonctions sont souvent présentes dans l'IAT des fichiers packés.

Dans les DLLs standards de l'installation de Windows, IAT es souvent situé juste après le début du fichier PE. Supposément, c'est fait par soucis d'optimisation.

Pendant le chargement, le fichier .exe n'est pas chargé dans la mémoire d'un seul coup (rappelez-vous ces fichiers d'installation énormes qui sont démarrés de façon douteusement rapide), ils sont «mappés», et chargés en mémoire par parties lorsqu'elles sont accédées.

Les développeurs de Microsoft ont sûrement décidé que ça serait plus rapide.

Ressources

Les ressources dans un fichier PE sont seulement un ensemble d'icônes, d'images, de chaînes de texte, de descriptions de boîtes de dialogues.

Peut-être qu'elles ont été séparées du code principal afin de pouvoir être multilingue, et il est plus simple de prendre du texte ou une image pour la langue qui est définie

[34] MSDN. Il y a aussi le «new-style binding».
[35] Aussi disponible en `http://msdn.microsoft.com/en-us/magazine/bb985992.aspx`

dans l'OS.

Par effet de bord, elles peuvent être éditées facilement et sauvées dans le fichier exécutable, même si on n'a pas de connaissance particulière, en utilisant l'éditeur ResHack, par exemple (6.5.2).

.NET

Les programmes .NET ne sont pas compilés en code machine, mais dans un bytecode spécial. Strictement parlant, il y a du bytecode à la place du code x86 usuel dans le fichier .eexe, toutefois, le point d'entrée (OEP) point sur un petit fragment de code x86:

```
jmp         mscoree.dll!_CorExeMain
```

Le loader de .NET se trouve dans mscoree.dll, qui traite le fichier PE.

Il en était ainsi dans tous les OSes pre-Windows XP. Á partir de XP, le loader de l'OS est capable de détecter les fichiers .NET et le lance sans exécuter cette instruction JMP[36].

TLS

Cette section contient les données initialisées pour le TLS(6.2 on page 965) (si nécessaire). Lorsque qu'une nouvelle thread démarre, ses données TLS sont initialisées en utilisant les données de cette section.

Á part ça, la spécification des fichiers PE fourni aussi l'initialisation de la section TLS, aussi appelée TLS callbacks.

Si elles sont présentes, elles doivent être appelées avant que le contrôle ne soit passé au point d'entrée principal (OEP).

Ceci est largement utilisé dans les packeurs/chiffreurs de fichiers PE.

Outils

- objdump (présent dans cygwin) pour afficher toutes les structures d'un fichier PE.
- Hiew(7.1 on page 1029) comme éditeur.
- pefile—bibliothèque-Python pour le traitement des fichiers PE [37].
- ResHack AKA Resource Hacker—éditeur de ressources[38].

[36] MSDN
[37] https://code.google.com/p/pefile/
[38] https://code.google.com/p/pefile/

- PE_add_import[39]—outil simple pour ajouter des symboles à la table d'imports d'un exécutable au format PE.
- PE_patcher[40]—outil simple pour patcher les exécutables PE.
- PE_search_str_refs[41]—outil simple pour chercher une fonction dans un exécutable PE qui utilise une certaine chaîne de texte.

Autres lectures

- Daniel Pistelli—The .NET File Format [42]

6.5.3 Windows SEH

Oublions MSVC

Sous Windows, SEH concerne la gestion d'exceptions. Ce mécanisme est indépendant du langage de programmation et n'est en aucun cas spécifique ni à C++, ni à l'POO.

Nous nous intéressons donc au SEH indépendamment du C++ et des extensions du langage dans MSVC.

À chaque processus est associé une chaîne de gestionnaires SEH. A tout moment, chaque TIB référence le gestionnaire le plus récent de la chaîne.

Dès qu'une exception intervient (division par zéro, violation d'accès mémoire, exception explicitement déclenchée par le programme en appelant la fonction `RaiseException()` ...), l'OS retrouve dans le TIB le gestionnaire le plus récemment déclaré et l'appelle. Il lui fourni l'état de la CPU (contenu des registres ...) tels qu'ils étaient lors du déclenchement de l'exception.

Le gestionnaire examine le type de l'exception et décide de la traiter s'il la reconnaît. Sinon, il signale à l'OS qu'il passe la main et celui-ci appelle le prochain gestionnaire dans la chaîne, et ainsi de suite jusqu'à ce qu'il trouve un gestionnaire qui soit capable de la traiter.

À la toute fin de la chaîne se trouve un gestionnaire standard qui affiche la fameuse boîte de dialogue qui informe l'utilisateur que le processus va être avorté. Ce dialogue fournit quelques informations techniques au sujet de l'état de la CPU au moment du crash, et propose de collecter des informations techniques qui seront envoyées aux développeurs de Microsoft.

[39] http://yurichev.com/PE_add_imports.html
[40] yurichev.com
[41] yurichev.com
[42] http://www.codeproject.com/Articles/12585/The-NET-File-Format

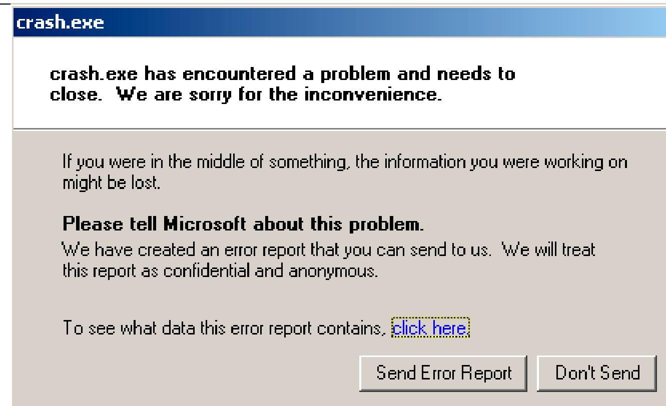


Fig. 6.2: Windows XP

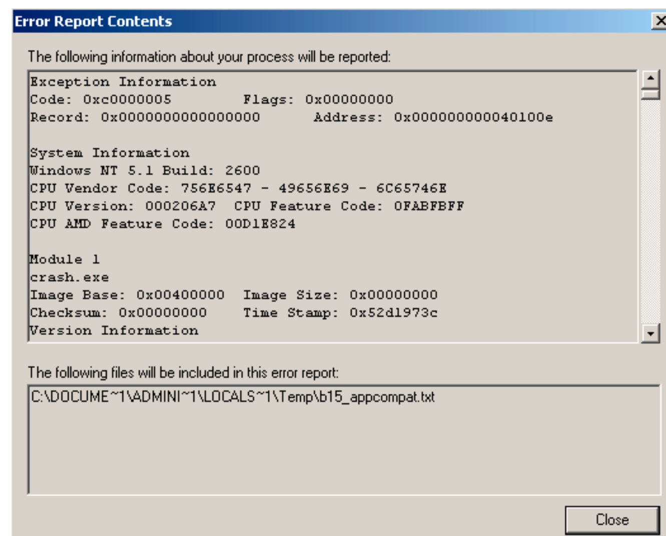


Fig. 6.3: Windows XP

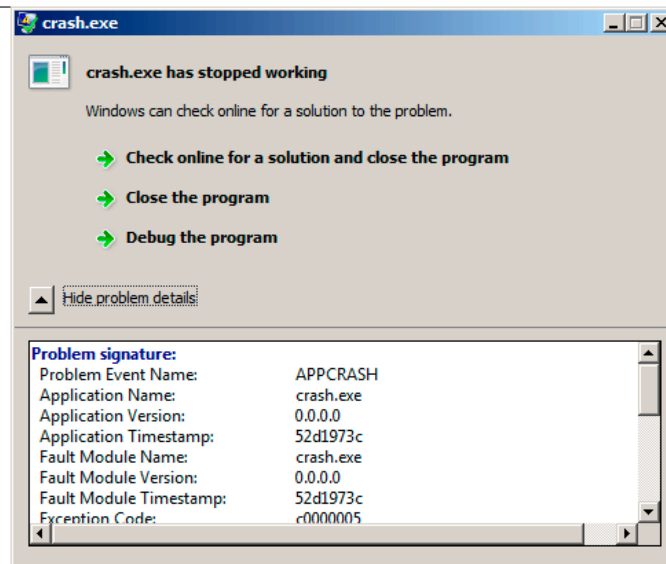


Fig. 6.4: Windows 7

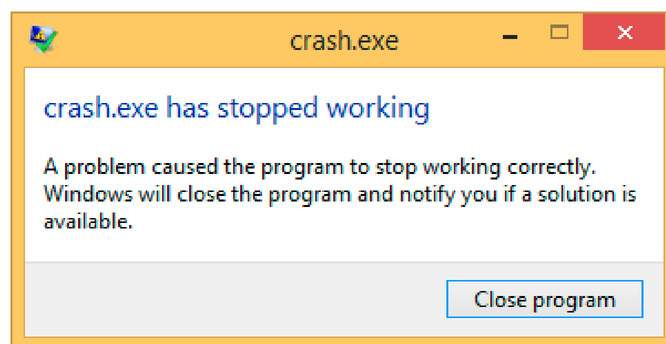


Fig. 6.5: Windows 8.1

Historiquement, cette chaîne de gestionnaires était connue sous l'appellation Dr. Watson [43].

Certains développeurs ont eu l'idée d'écrire leur propre gestionnaire d'exceptions pour recevoir les informations relatives au crash. Ils enregistrent leur gestionnaire en appelant la fonction `SetUnhandledExceptionFilter()` qui sera alors appelée si l'OS ne trouve aucun autre gestionnaire qui souhaite gérer l'exception.

Oracle RDBMS— en est un bon exemple qui sauvegarde un énorme fichier dump collectant toutes les informations possibles concernant la CPU et l'état de la mémoire.

Écrivons notre propre gestionnaire d'exception. Cet exemple s'appuie sur celui de [Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Hand-*

[43] Wikipédia

ling, (1997)][44]. Pour le compiler, il faut utiliser l'option SAFESEH : `cl seh1.cpp /link /safeseh:no`. Vous trouverez plus d'informations concernant SAFESEH à: MSDN.

```
#include <windows.h>
#include <stdio.h>

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
                struct _EXCEPTION_RECORD *ExceptionRecord,
                void * EstablisherFrame,
                struct _CONTEXT *ContextRecord,
                void * DispatcherContext )
{
        unsigned i;

        printf ("%s\n", __FUNCTION__);
        printf ("ExceptionRecord->ExceptionCode=0x%p\n", ExceptionRecord->↙
    ↳ ExceptionCode);
        printf ("ExceptionRecord->ExceptionFlags=0x%p\n", ExceptionRecord->↙
    ↳ ExceptionFlags);
        printf ("ExceptionRecord->ExceptionAddress=0x%p\n", ExceptionRecord↙
    ↳ ->ExceptionAddress);

        if (ExceptionRecord->ExceptionCode==0xE1223344)
        {
                printf ("That's for us\n");
                // yes, we "handled" the exception
                return ExceptionContinueExecution;
        }
        else if (ExceptionRecord->ExceptionCode==EXCEPTION_ACCESS_VIOLATION↙
    ↳ )
        {
                printf ("ContextRecord->Eax=0x%08X\n", ContextRecord->Eax);
                // will it be possible to 'fix' it?
                printf ("Trying to fix wrong pointer address\n");
                ContextRecord->Eax=(DWORD)&new_value;
                // yes, we "handled" the exception
                return ExceptionContinueExecution;
        }
        else
        {
                printf ("We do not handle this\n");
                // someone else's problem
                return ExceptionContinueSearch;
        };
}

int main()
{
        DWORD handler = (DWORD)except_handler; // take a pointer to our
    handler
```

[44]Aussi disponible en http://www.microsoft.com/msj/0197/Exception/Exception.aspx

```
        // install exception handler
        __asm
        {                               // make EXCEPTION_REGISTRATION
    record:
                push    handler         // address of handler function
                push    FS:[0]          // address of previous handler
                mov     FS:[0],ESP      // add new EXECEPTION_REGISTRATION
        }

        RaiseException (0xE1223344, 0, 0, NULL);

        // now do something very bad
        int* ptr=NULL;
        int val=0;
        val=*ptr;
        printf ("val=%d\n", val);

        // deinstall exception handler
        __asm
        {                               // remove our EXECEPTION_REGISTRATION
    record
                mov     eax,[ESP]       // get pointer to previous record
                mov     FS:[0], EAX     // install previous record
                add     esp, 8          // clean our EXECEPTION_REGISTRATION
    off stack
        }

        return 0;
}
```

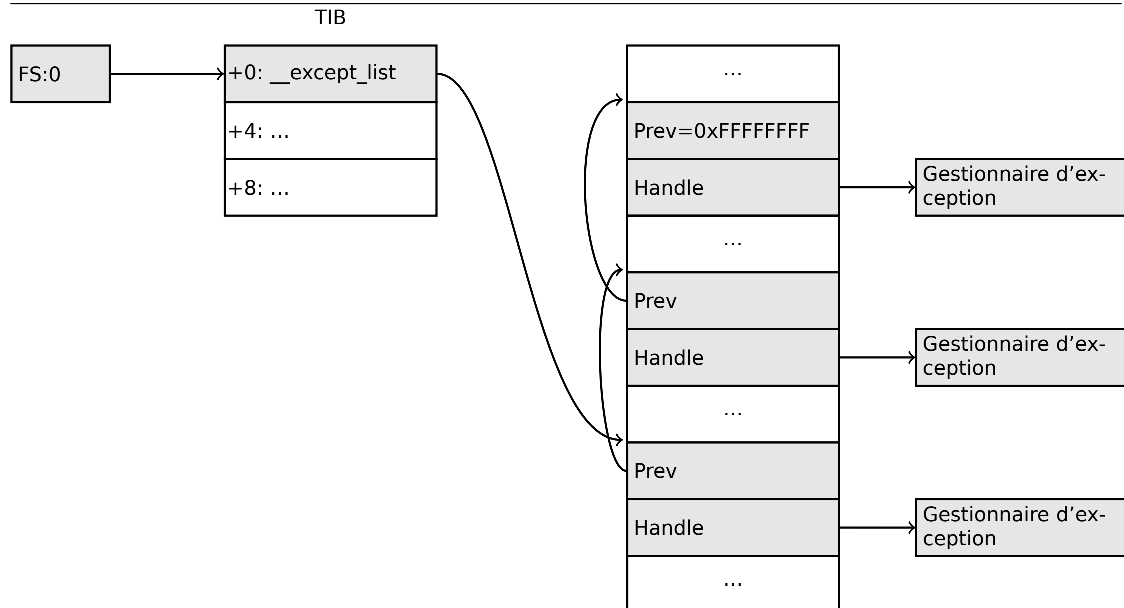
En environnement win32, le registre de segment FS: contient l'adresse du TIB.

Le tout premier élément de la structure TIB est un pointeur sur le premier gestionnaire de la chaîne de traitement des exceptions. Nous le sauvegardons sur la pile et remplaçons la valeur par celle de notre propre gestionnaire. La structure est du type _EXCEPTION_REGISTRATION. Il s'agit d'une simple liste chaînée dont les éléments sont conservés sur la pile.

Listing 6.22 : MSVC/VC/crt/src/exsup.inc

```
_EXCEPTION_REGISTRATION struc
     prev    dd      ?
     handler dd      ?
_EXCEPTION_REGISTRATION ends
```

Le champ «handler» contient l'adresse du gestionnaire et le champ «prev» celle de l'enregistrement suivant dans la chaîne. Le dernier enregistrement contient la valeur 0xFFFFFFFF (-1) dans son champ «prev».



Une fois notre gestionnaire installé, nous invoquons la fonction `RaiseException()` [45]. Il s'agit d'une exception utilisateur. Le gestionnaire vérifie le code. Si le code est égal à `0xE1223344`, il retourne la valeur `ExceptionContinueExecution` qui signifie que le gestionnaire a corrigé le contenu de la structure passée en paramètre qui décrit l'état de la CPU. La modification concerne généralement les registres EIP/ESP. L'OS peut alors reprendre l'exécution du thread.

Si vous modifiez légèrement le code pour que le gestionnaire retourne la valeur `ExceptionContinueSearch`, l'OS appellera les gestionnaires suivants dans la liste. Il est peu probable que l'un d'eux sache la gérer puisqu'aucun d'eux ne la comprend, ni ne connaît le code exception. Vous verrez donc apparaître la boîte de dialogue Windows précurseur du crash.

Quelles sont les différences entre les exceptions système et les exceptions utilisateur ? Les exceptions système sont listées ci-dessous:

[45] MSDN

as defined in WinBase.h	as defined in ntstatus.h	value
EXCEPTION_ACCESS_VIOLATION	STATUS_ACCESS_VIOLATION	0xC0000005
EXCEPTION_DATATYPE_MISALIGNMENT	STATUS_DATATYPE_MISALIGNMENT	0x80000002
EXCEPTION_BREAKPOINT	STATUS_BREAKPOINT	0x80000003
EXCEPTION_SINGLE_STEP	STATUS_SINGLE_STEP	0x80000004
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	STATUS_ARRAY_BOUNDS_EXCEEDED	0xC000008C
EXCEPTION_FLT_DENORMAL_OPERAND	STATUS_FLOAT_DENORMAL_OPERAND	0xC000008D
EXCEPTION_FLT_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_BY_ZERO	0xC000008E
EXCEPTION_FLT_INEXACT_RESULT	STATUS_FLOAT_INEXACT_RESULT	0xC000008F
EXCEPTION_FLT_INVALID_OPERATION	STATUS_FLOAT_INVALID_OPERATION	0xC0000090
EXCEPTION_FLT_OVERFLOW	STATUS_FLOAT_OVERFLOW	0xC0000091
EXCEPTION_FLT_STACK_CHECK	STATUS_FLOAT_STACK_CHECK	0xC0000092
EXCEPTION_FLT_UNDERFLOW	STATUS_FLOAT_UNDERFLOW	0xC0000093
EXCEPTION_INT_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVIDE_BY_ZERO	0xC0000094
EXCEPTION_INT_OVERFLOW	STATUS_INTEGER_OVERFLOW	0xC0000095
EXCEPTION_PRIV_INSTRUCTION	STATUS_PRIVILEGED_INSTRUCTION	0xC0000096
EXCEPTION_IN_PAGE_ERROR	STATUS_IN_PAGE_ERROR	0xC0000006
EXCEPTION_ILLEGAL_INSTRUCTION	STATUS_ILLEGAL_INSTRUCTION	0xC000001D
EXCEPTION_NONCONTINUABLE_EXCEPTION	STATUS_NONCONTINUABLE_EXCEPTION	0xC0000025
EXCEPTION_STACK_OVERFLOW	STATUS_STACK_OVERFLOW	0xC00000FD
EXCEPTION_INVALID_DISPOSITION	STATUS_INVALID_DISPOSITION	0xC0000026
EXCEPTION_GUARD_PAGE	STATUS_GUARD_PAGE_VIOLATION	0x80000001
EXCEPTION_INVALID_HANDLE	STATUS_INVALID_HANDLE	0xC0000008
EXCEPTION_POSSIBLE_DEADLOCK	STATUS_POSSIBLE_DEADLOCK	0xC0000194
CONTROL_C_EXIT	STATUS_CONTROL_C_EXIT	0xC000013A

Le code de chaque exception se décompose comme suit:

31	29	28	27 … 16	15 … 0
S	U	0	Facility code	Error code

S est un code status de base: 11—erreur; 10—warning; 01—information; 00—succès. U—lorsqu'il s'agit d'un code utilisateur.

Voici pourquoi nous choisissons le code 0xE1223344—E_{16} (1110_2) 0xE (1110b) qui signifie 1) qu'il s'agit d'une exception utilisateur; 2) qu'il s'agit d'une erreur.

Pour être honnête, l'exemple fonctionne aussi bien sans ces bits de poids fort.

Tentons maintenant de lire la valeur à l'adresse mémoire 0.

Bien entendu, dans win32 il n'existe rien à cette adresse, ce qui déclenche une exception.

Le premier gestionnaire à être invoqué est le vôtre. Il reconnaît l'exception car il compare le code avec celui de la constante `EXCEPTION_ACCESS_VIOLATION`.

Le code qui lit la mémoire à l'adresse 0 ressemble à ceci:

Listing 6.23 : MSVC 2010

```
        ...
        xor     eax, eax
        mov     eax, DWORD PTR [eax] ; exception will occur here
        push    eax
        push    OFFSET msg
```

```
        call    _printf
        add     esp, 8
        ...
```

Serait-il possible de corriger cette erreur «au vol» afin de continuer l'exécution du programme?

Notre gestionnaire d'exception peut modifier la valeur du registre EAX puis laisser l'OS exécuter de nouveau l'instruction fautive. C'est ce que nous faisons et la raison pour laquelle `printf()` affiche 1234. Lorsque notre gestionnaire a fini son travail, la valeur de EAX n'est plus 0 mais l'adresse de la variable globale `new_value`. L'exécution du programme se poursuit donc.

Voici ce qui se passe: le gestionnaire mémoire de la CPU signale une erreur, la CPU suspend le thread, trouve le gestionnaire d'exception dans le noyau Windows, lequel à son tour appelle les gestionnaires de la chaîne SEH un par un.

Nous utilisons ici le compilateur MSVC 2010. Bien entendu, il n'y a aucune garantie que celui-ci décide d'utiliser le registre EAX pour conserver la valeur du pointeur.

Le truc du remplacement du contenu du registre n'est qu'une illustration de ce que peut être le fonctionnement interne des SEH. En pratique, il est très rare qu'il soit utilisé pour corriger «on-the-fly» une erreur.

Pourquoi les enregistrements SEH sont-ils conservés directement sur la pile et non pas à un autre endroit?

L'explication la plus plausible est que l'OS n'a ainsi pas besoin de libérer l'espace qu'ils utilisent. Ces enregistrements sont automatiquement supprimés lorsque la fonction se termine. C'est un peu comme la fonction alloca() : (1.9.2 on page 49).

Retour à MSVC

Microsoft a ajouté un mécanisme non standard de gestion d'exceptions à MSVC[46] essentiellement à l'usage des programmeurs C. Ce mécanisme est totalement distinct de celui définit par le standard ISO du langage C++.

```
__try
{
    ...
}
__except(filter code)
{
    handler code
}
```

À la place du gestionnaire d'exception, on peut trouver un block «Finally» :

```
__try
{
    ...
}
```

[46] MSDN

```
__finally
{
    ...
}
```

Le code de filtrage est une expression. L'évaluation de celle-ci permet de définir si le gestionnaire reconnaît l'exception qui a été déclenchée.

Si votre filtre est trop complexe pour tenir dans une seule expression, une fonction de filtrage séparée peut être définie.

Il existe de nombreuses constructions de ce type dans le noyau Windows. En voici quelques exemples (WRK) :

Listing 6.24 : WRK-v1.2/base/ntos/ob/obwait.c

```
try {

    KeReleaseMutant( (PKMUTANT)SignalObject,
                     MUTANT_INCREMENT,
                     FALSE,
                     TRUE );

} except((GetExceptionCode () == STATUS_ABANDONED ||
          GetExceptionCode () == STATUS_MUTANT_NOT_OWNED)?
             EXCEPTION_EXECUTE_HANDLER :
             EXCEPTION_CONTINUE_SEARCH) {
    Status = GetExceptionCode();

    goto WaitExit;
}
```

Listing 6.25 : WRK-v1.2/base/ntos/cache/cachesub.c

```
try {

    RtlCopyBytes( (PVOID)((PCHAR)CacheBuffer + PageOffset),
                  UserBuffer,
                  MorePages ?
                    (PAGE_SIZE - PageOffset) :
                    (ReceivedLength - PageOffset) );

} except( CcCopyReadExceptionFilter( GetExceptionInformation(),
                                                             &Status ) ) {
```

Voici aussi un exemple de code de filtrage:

Listing 6.26 : WRK-v1.2/base/ntos/cache/copysup.c

```
LONG
CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer,
    IN PNTSTATUS ExceptionCode
    )
```

```
/*++

Routine Description:

    This routine serves as an exception filter and has the special job of
    extracting the "real" I/O error when Mm raises STATUS_IN_PAGE_ERROR
    beneath us.

Arguments:

    ExceptionPointer - A pointer to the exception record that contains
                       the real Io Status.

    ExceptionCode - A pointer to an NTSTATUS that is to receive the real
                    status.

Return Value:

    EXCEPTION_EXECUTE_HANDLER

--*/

{
    *ExceptionCode = ExceptionPointer->ExceptionRecord->ExceptionCode;

    if ( (*ExceptionCode == STATUS_IN_PAGE_ERROR) &&
         (ExceptionPointer->ExceptionRecord->NumberParameters >= 3) ) {

        *ExceptionCode = (NTSTATUS) ExceptionPointer->ExceptionRecord->⤦
    ↳ ExceptionInformation[2];
    }

    ASSERT( !NT_SUCCESS(*ExceptionCode) );

    return EXCEPTION_EXECUTE_HANDLER;
}
```

En interne, SEH est une extension du mécanisme de gestion des exceptions implémenté par l'OS. La fonction de gestion d'exceptions est `_except_handler3` (pour SEH3) ou `_except_handler4` (pour SEH4).

Le code de ce gestionnaire est propre à MSVC et est situé dans ses librairies, ou dans msvcr*.dll. Il est essentiel de comprendre que SEH est purement lié à MSVC.

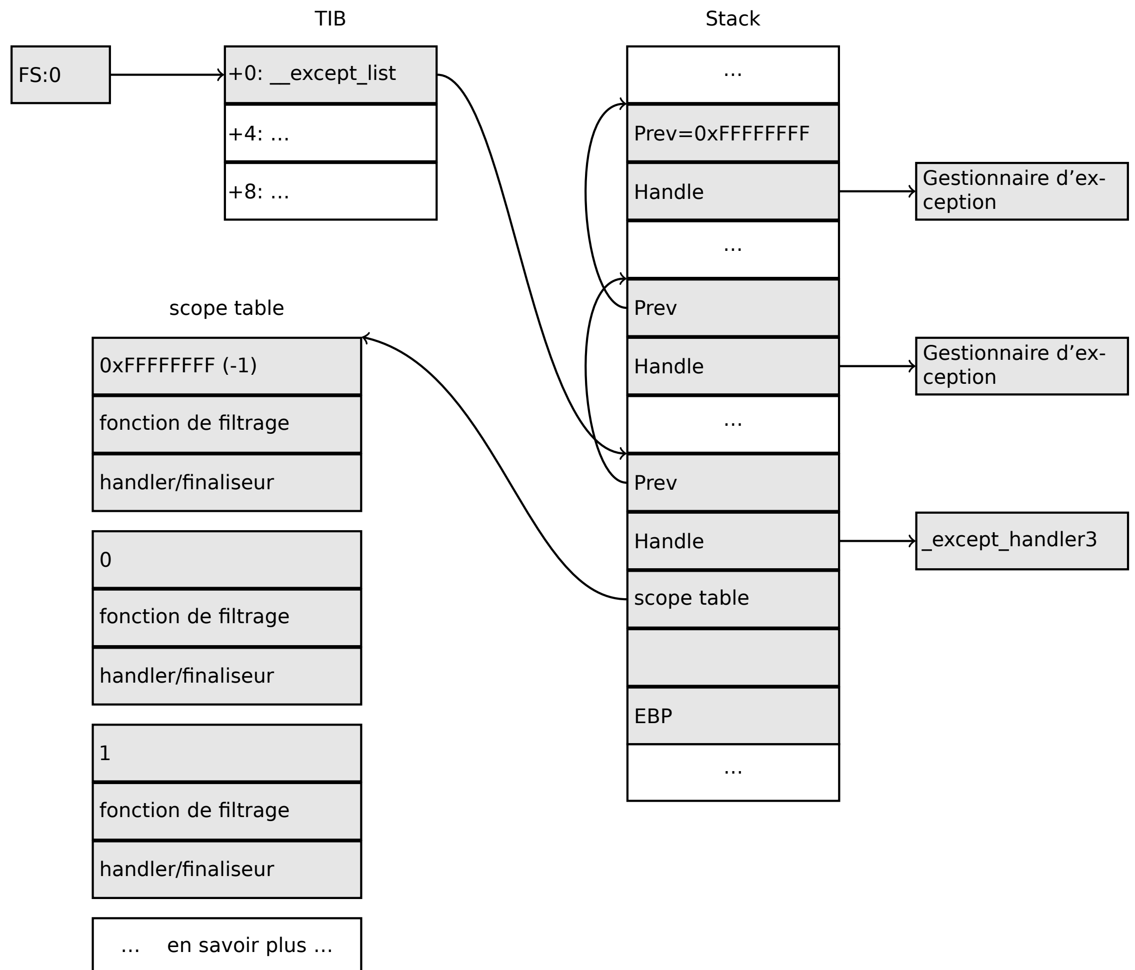
D'autres compilateurs win32 peuvent choisir un modèle totalement différent.

SEH3

SEH3 est géré par la fonction `_except_handler3`. Il ajoute à la structure `_EXCEPTION_REGISTRATION` un pointeur vers une *scope table* et une variable *previous try level*. SEH4 de son côté ajoute 4 valeurs à la structure *scope table* pour la gestion des dépassements de

buffer.

La structure *scope table* est un ensemble de pointeurs vers les blocs de code du filtre et du gestionnaire de chaque niveau *try/except* imbriqué.



Il est essentiel de comprendre que l'OS ne se préoccupe que des champs *prev/handle* et de rien d'autre.
Les autres champs sont exploités par la fonction `_except_handler3`, de même que le contenu de la structure *scope table* afin de décider quel gestionnaire exécuter et quand.

Le code source de la fonction `_except_handler3` n'est pas public.

Cependant, le système d'exploitation Sanos, possède un mode de compatibilité win32. Celui-ci ré-implémente les mêmes fonctions d'une manière quasi équivalente à celle de Windows [47]. On trouve une autre ré-implémentation dans Wine[48] ainsi que dans

[47] `https://code.google.com/p/sanos/source/browse/src/win32/msvcrt/except.c`
[48] GitHub

ReactOS[49].

Lorsque le champ *filter* est un pointeur NULL, le champ *handler* est un pointeur vers un bloc de code *finally*.

Au cours de l'exécution, la valeur du champ *previous try level* change.
Ceci permet à la fonction `_except_handler3` de connaître le niveau d'imbrication et donc de savoir quelle entrée de la table *scope table* utiliser en cas d'exception.

SEH3: exemple de bloc try/except

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int main()
{
    int* p = NULL;
    __try
    {
        printf("hello #1!\n");
        *p = 13;    // causes an access violation exception;
        printf("hello #2!\n");
    }
    __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
            EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        printf("access violation, can't recover\n");
    }
}
```

Listing 6.27 : MSVC 2003

```
$SG74605 DB    'hello #1!', 0aH, 00H
$SG74606 DB    'hello #2!', 0aH, 00H
$SG74608 DB    'access violation, can''t recover', 0aH, 00H
_DATA    ENDS

; scope table:
CONST     SEGMENT
$T74622   DD    0ffffffffH    ; previous try level
          DD    FLAT:$L74617  ; filter
          DD    FLAT:$L74618  ; handler

CONST    ENDS
_TEXT    SEGMENT
$T74621 = -32 ; size = 4
_p$ = -28     ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC NEAR
```

[49] http://doxygen.reactos.org/d4/df2/lib_2sdk_2crt_2except_2except_8c_source.html

```
    push   ebp
    mov    ebp, esp
    push   -1                            ; previous try level
    push   OFFSET FLAT:$T74622           ; scope table
    push   OFFSET FLAT:__except_handler3   ; handler
    mov    eax, DWORD PTR fs:__except_list
    push   eax                           ; prev
    mov    DWORD PTR fs:__except_list, esp
    add    esp, -16
; 3 registers to be saved:
    push   ebx
    push   esi
    push   edi
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0  ; previous try level
    push   OFFSET FLAT:$SG74605 ; 'hello #1!'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    push   OFFSET FLAT:$SG74606 ; 'hello #2!'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1  ; previous try level
    jmp    SHORT $L74616

    ; filter code:
$L74617:
$L74627:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74621[ebp], eax
    mov    eax, DWORD PTR $T74621[ebp]
    sub    eax, -1073741819; c0000005H
    neg    eax
    sbb    eax, eax
    inc    eax
$L74619:
$L74626:
    ret    0

    ; handler code:
$L74618:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74608 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; setting previous try level back
    to -1
$L74616:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
```

```
    mov    DWORD PTR fs:__except_list, ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

Nous voyons ici la manière dont le bloc SEH est construit sur la pile. La structure *scope table* est présente dans le segment CONST du programme— ce qui est normal puisque son contenu n'a jamais besoin d'être changé. Un point intéressant est la manière dont la valeur de la variable *previous try level* évolue. Sa valeur initiale est 0xFFFFFFFF (-1). L'entrée dans le bloc try débute par l'écriture de la valeur 0 dans la variable. La sortie du bloc try est marquée par la restauration de la valeur -1. Nous voyons également l'adresse du bloc de filtrage et de celui du gestionnaire.

Nous pouvons donc observer facilement la présence de blocs *try/except* dans la fonction.

Le code d'initialisation des structures SEH dans le prologue de la fonction peut être partagé par de nombreuses fonctions. Le compilateur choisi donc parfois d'insérer dans le prologue d'une fonction un appel à la fonction SEH_prolog() qui assure cette initialisation.

Le code de nettoyage des structures SEH se trouve quant à lui dans la fonction SEH_epilog().

Tentons d'exécuter cet exemple dans tracer :

```
tracer.exe -l:2.exe --dump-seh
```

Listing 6.28 : tracer.exe output

```
EXCEPTION_ACCESS_VIOLATION at 2.exe!main+0x44 (0x401054) ↙
    ↘ ExceptionInformation[0]=1
EAX=0x00000000 EBX=0x7efde000 ECX=0x0040cbc8 EDX=0x0008e3c8
ESI=0x00001db1 EDI=0x00000000 EBP=0x0018feac ESP=0x0018fe80
EIP=0x00401054
FLAGS=AF IF RF
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x401204 (2.exe!↙
    ↘ _except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401070 (2.exe!main+0↙
    ↘ x60) handler=0x401088 (2.exe!main+0x78)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x401204 (2.exe!↙
    ↘ _except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401531 (2.exe!↙
    ↘ mainCRTStartup+0x18d) handler=0x401545 (2.exe!mainCRTStartup+0x1a1)
```

```
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!↙
    ↳ __except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header:    GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
                EHCookieOffset=0xffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!↙
    ↳ ___safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!↙
    ↳ _TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!↙
    ↳ _FinalExceptionHandler@16)
```

Nous constatons que la chaîne SEH est constituée de 4 gestionnaires.

Le deux premiers sont situés dans le code de notre exemple. Deux? Mais nous n'en avons défini qu'un! Effectivement, mais un second a été initialisé dans la fonction `_mainCRTStartup()` du CRT. Il semble que celui-ci gère au moins les exceptions FPU. Son code source figure dans le fichier `crt/src/winxfltr.c` fournit avec l'installation de MSVC.

Le troisième est le gestionnaire SEH4 dans ntdll.dll. Le quatrième n'est pas lié à MSVC et se situe dans ntdll.dll. Son nom suffit à en décrire l'utilité.

Comme vous le constatez, nous avons 3 types de gestionnaire dans la même chaîne:

L'un n'a rien à voir avec MSVC (le dernier) et deux autres sont liés à MSVC: SEH3 et SEH4.

SEH3: exemple de deux blocs try/except

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int filter_user_exceptions (unsigned int code, struct _EXCEPTION_POINTERS *↙
    ↳ ep)
{
    printf("in filter. code=0x%08X\n", code);
    if (code == 0x112233)
    {
        printf("yes, that is our exception\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        printf("not our exception\n");
        return EXCEPTION_CONTINUE_SEARCH;
    };
}
int main()
{
    int* p = NULL;
```

```
    __try
    {
        __try
        {
            printf ("hello!\n");
            RaiseException (0x112233, 0, 0, NULL);
            printf ("0x112233 raised. now let's crash\n");
            *p = 13;    // causes an access violation exception;
        }
        __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
                EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
        {
            printf("access violation, can't recover\n");
        }
    }
    __except(filter_user_exceptions(GetExceptionCode(), ↙
    ↳ GetExceptionInformation()))
    {
        // the filter_user_exceptions() function answering to the question
        // "is this exception belongs to this block?"
        // if yes, do the follow:
        printf("user exception caught\n");
    }
}
```

Nous avons maintenant deux blocs `try`. La structure *scope table* possède donc deux entrées, une pour chaque bloc. La valeur de *Previous try level* change selon que l'exécution entre ou sort des blocs `try`.

Listing 6.29 : MSVC 2003

```
$SG74606 DB    'in filter. code=0x%08X', 0aH, 00H
$SG74608 DB    'yes, that is our exception', 0aH, 00H
$SG74610 DB    'not our exception', 0aH, 00H
$SG74617 DB    'hello!', 0aH, 00H
$SG74619 DB    '0x112233 raised. now let''s crash', 0aH, 00H
$SG74621 DB    'access violation, can''t recover', 0aH, 00H
$SG74623 DB    'user exception caught', 0aH, 00H

_code$ = 8   ; size = 4
_ep$ = 12    ; size = 4
_filter_user_exceptions PROC NEAR
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _code$[ebp]
    push   eax
    push   OFFSET FLAT:$SG74606 ; 'in filter. code=0x%08X'
    call   _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867; 00112233H
    jne    SHORT $L74607
    push   OFFSET FLAT:$SG74608 ; 'yes, that is our exception'
    call   _printf
    add    esp, 4
```

```
    mov    eax, 1
    jmp    SHORT $L74605
$L74607:
    push   OFFSET FLAT:$SG74610 ; 'not our exception'
    call   _printf
    add    esp, 4
    xor    eax, eax
$L74605:
    pop    ebp
    ret    0
_filter_user_exceptions ENDP

; scope table:
CONST    SEGMENT
$T74644   DD    0ffffffffH   ; previous try level for outer block
          DD    FLAT:$L74634 ; outer block filter
          DD    FLAT:$L74635 ; outer block handler
          DD    00H          ; previous try level for inner block
          DD    FLAT:$L74638 ; inner block filter
          DD    FLAT:$L74639 ; inner block handler
CONST    ENDS

$T74643 = -36     ; size = 4
$T74642 = -32     ; size = 4
_p$ = -28         ; size = 4
__$SEHRec$ = -24  ; size = 24
_main    PROC NEAR
    push   ebp
    mov    ebp, esp
    push   -1 ; previous try level
    push   OFFSET FLAT:$T74644
    push   OFFSET FLAT:__except_handler3
    mov    eax, DWORD PTR fs:__except_list
    push   eax
    mov    DWORD PTR fs:__except_list, esp
    add    esp, -20
    push   ebx
    push   esi
    push   edi
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; outer try block entered. set
    previous try level to 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 1 ; inner try block entered. set
    previous try level to 1
    push   OFFSET FLAT:$SG74617 ; 'hello!'
    call   _printf
    add    esp, 4
    push   0
    push   0
    push   0
    push   1122867    ; 00112233H
    call   DWORD PTR __imp__RaiseException@16
    push   OFFSET FLAT:$SG74619 ; '0x112233 raised. now let''s crash'
```

```
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set
    previous try level back to 0
    jmp    SHORT $L74615

; inner block filter:
$L74638:
$L74650:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74643[ebp], eax
    mov    eax, DWORD PTR $T74643[ebp]
    sub    eax, -1073741819; c0000005H
    neg    eax
    sbb    eax, eax
    inc    eax
$L74640:
$L74648:
    ret    0

; inner block handler:
$L74639:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74621 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set
    previous try level back to 0

$L74615:
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; outer try block exited, set
    previous try level back to -1
    jmp    SHORT $L74633

; outer block filter:
$L74634:
$L74651:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74642[ebp], eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    push   ecx
    mov    edx, DWORD PTR $T74642[ebp]
    push   edx
    call   _filter_user_exceptions
    add    esp, 8
$L74636:
$L74649:
    ret    0
```

```
; outer block handler:
$L74635:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74623 ; 'user exception caught'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; both try blocks exited. set
    previous try level back to -1
$L74633:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:__except_list, ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Si nous positionnons un point d'arrêt sur la fonction `printf()` qui est appelée par le gestionnaire, nous pouvons constater comment un nouveau gestionnaire SEH est ajouté.

Il s'agit peut-être d'un autre mécanisme interne de la gestion SEH. Nous constatons aussi que notre structure *scope table* contient 2 entrées.

```
tracer.exe -l:3.exe bpx=3.exe!printf --dump-seh
```

Listing 6.30 : tracer.exe output

```
(0) 3.exe!printf
EAX=0x0000001b EBX=0x00000000 ECX=0x0040cc58 EDX=0x0008e3c8
ESI=0x00000000 EDI=0x00000000 EBP=0x0018f840 ESP=0x0018f838
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18fe9c handler=0x771db4ad (ntdll.dll!↙
    ↘ ExecuteHandler2@20+0x3a)
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x4012e0 (3.exe!↙
    ↘ _except_handler3)
SEH3 frame. previous trylevel=1
scopetable entry[0]. previous try level=-1, filter=0x401120 (3.exe!main+0↙
    ↘ xb0) handler=0x40113b (3.exe!main+0xcb)
scopetable entry[1]. previous try level=0, filter=0x4010e8 (3.exe!main+0x78↙
    ↘ ) handler=0x401100 (3.exe!main+0x90)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x4012e0 (3.exe!↙
    ↘ _except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x40160d (3.exe!↙
    ↘ mainCRTStartup+0x18d) handler=0x401621 (3.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!↙
    ↘ __except_handler4)
SEH4 frame. previous trylevel=0
```

```
SEH4 header:     GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
                 EHCookieOffset=0xffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!↙
    ↳ ___safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!↙
    ↳ _TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!↙
    ↳ _FinalExceptionHandler@16)
```

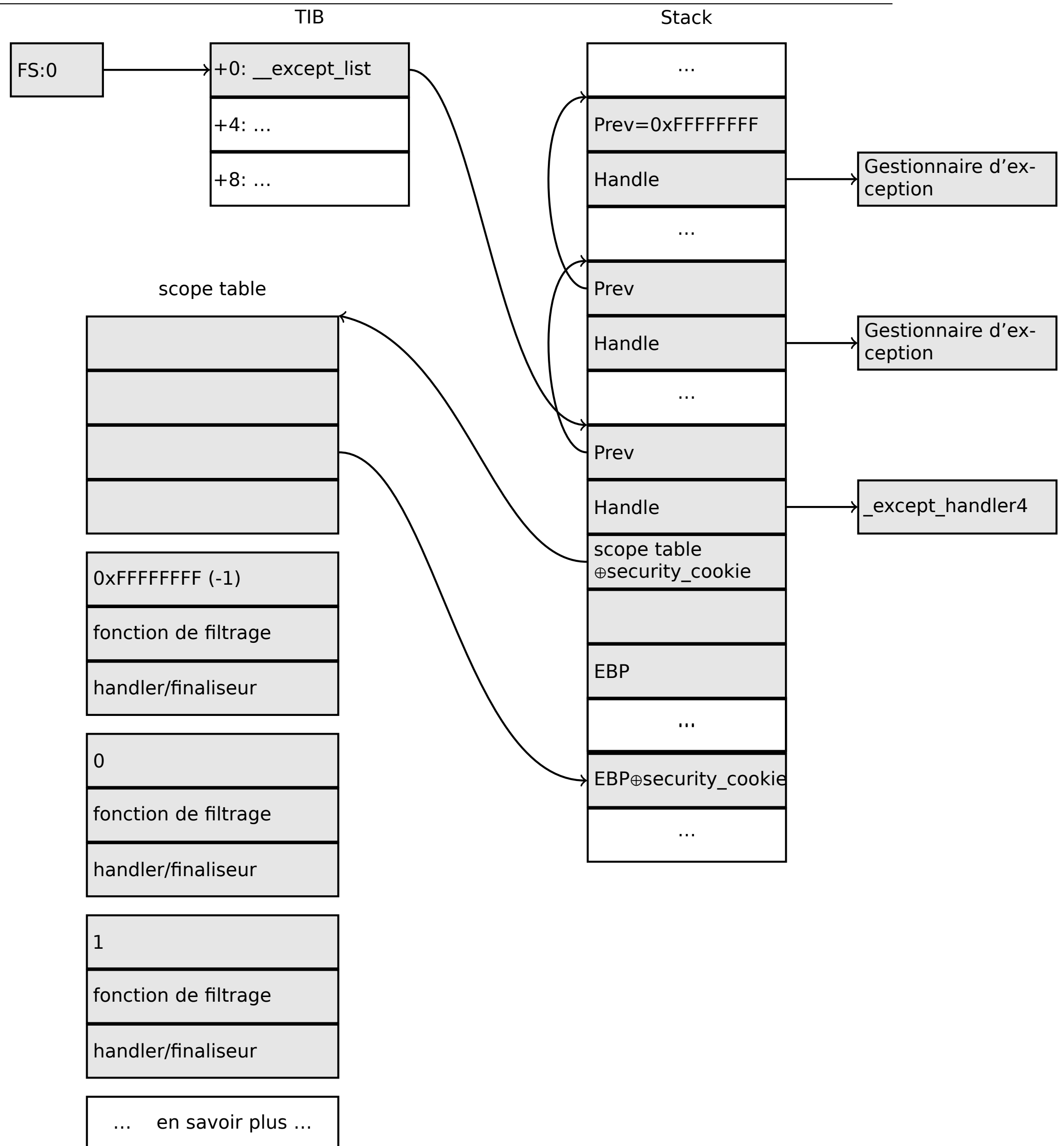
SEH4

Lors d'une attaque par dépassement de buffer (1.26.2 on page 350), l'adresse de la *scope table* peut être modifiée. C'est pourquoi à partir de MSVC 2005 SEH3 a été amélioré vers SEH4 pour ajouter une protection contre ce type d'attaque. Le pointeur vers la structure *scope table* est désormais xored avec la valeur d'un security cookie. Par ailleurs, la structure *scope table* a été étendue avec une en-tête contenant deux pointeurs vers des *security cookies*.

Chaque élément contient un offset dans la pile d'une valeur correspondant à: adresse du stack frame (EBP) xored avec la valeur du `security_cookie` lui aussi situé sur la pile.

Durant la gestion d'exception, l'intégrité de cette valeur est vérifiée. La valeur de chaque *security cookie* situé sur la pile est aléatoire. Une attaque à distance ne pourra donc pas la deviner.

Avec SEH4, la valeur initiale de *previous try level* est de -2 et non de -1.



Voici deux exemples compilés avec MSVC 2012 et SEH4:

Listing 6.31 : MSVC 2012: exemple bloc try unique

```
$SG85485 DB    'hello #1!', 0aH, 00H
$SG85486 DB    'hello #2!', 0aH, 00H
$SG85488 DB    'access violation, can''t recover', 0aH, 00H

; scope table:
xdata$x          SEGMENT
__sehtable$_main DD 0fffffffeH   ; GS Cookie Offset
    DD           00H             ; GS Cookie XOR Offset
```

```
    DD          0ffffffccH      ; EH Cookie Offset
    DD          00H             ; EH Cookie XOR Offset
    DD          0fffffffeH      ; previous try level
    DD          FLAT:$LN12@main ; filter
    DD          FLAT:$LN8@main  ; handler
xdata$x         ENDS

$T2 = -36         ; size = 4
_p$ = -32         ; size = 4
tv68 = -28        ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC
    push   ebp
    mov    ebp, esp
    push   -2
    push   OFFSET __sehtable$_main
    push   OFFSET __except_handler4
    mov    eax, DWORD PTR fs:0
    push   eax
    add    esp, -20
    push   ebx
    push   esi
    push   edi
    mov    eax, DWORD PTR ___security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax ; xored pointer to scope table
    xor    eax, ebp
    push   eax                              ; ebp ^ security_cookie
    lea    eax, DWORD PTR __$SEHRec$[ebp+8] ;
    pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov    DWORD PTR fs:0, eax
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
    push   OFFSET $SG85485 ; 'hello #1!'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    push   OFFSET $SG85486 ; 'hello #2!'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
    jmp    SHORT $LN6@main

; filter:
$LN7@main:
$LN12@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T2[ebp], eax
    cmp    DWORD PTR $T2[ebp], -1073741819 ; c0000005H
    jne    SHORT $LN4@main
    mov    DWORD PTR tv68[ebp], 1
```

```
    jmp    SHORT $LN5@main
$LN4@main:
    mov    DWORD PTR tv68[ebp], 0
$LN5@main:
    mov    eax, DWORD PTR tv68[ebp]
$LN9@main:
$LN11@main:
    ret    0

; handler:
$LN8@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85488 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
$LN6@main:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:0, ecx
    pop    ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Listing 6.32 : MSVC 2012: exemple de deux blocs try

```
$SG85486 DB    'in filter. code=0x%08X', 0aH, 00H
$SG85488 DB    'yes, that is our exception', 0aH, 00H
$SG85490 DB    'not our exception', 0aH, 00H
$SG85497 DB    'hello!', 0aH, 00H
$SG85499 DB    '0x112233 raised. now let''s crash', 0aH, 00H
$SG85501 DB    'access violation, can''t recover', 0aH, 00H
$SG85503 DB    'user exception caught', 0aH, 00H

xdata$x    SEGMENT
__sehtable$_main DD 0fffffffeH        ; GS Cookie Offset
                 DD    00H             ; GS Cookie XOR Offset
                 DD    0ffffffc8H      ; EH Cookie Offset
                 DD    00H             ; EH Cookie Offset
                 DD    0fffffffeH      ; previous try level for outer block
                 DD    FLAT:$LN19@main ; outer block filter
                 DD    FLAT:$LN9@main  ; outer block handler
                 DD    00H             ; previous try level for inner block
                 DD    FLAT:$LN18@main ; inner block filter
                 DD    FLAT:$LN13@main ; inner block handler
xdata$x    ENDS

$T2 = -40        ; size = 4
$T3 = -36        ; size = 4
```

```
_p$ = -32        ; size = 4
tv72 = -28       ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC
    push   ebp
    mov    ebp, esp
    push   -2  ; initial previous try level
    push   OFFSET __sehtable$_main
    push   OFFSET __except_handler4
    mov    eax, DWORD PTR fs:0
    push   eax ; prev
    add    esp, -24
    push   ebx
    push   esi
    push   edi
    mov    eax, DWORD PTR ___security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax       ; xored pointer to scope
    table
    xor    eax, ebp                                ; ebp ^ security_cookie
    push   eax
    lea    eax, DWORD PTR __$SEHRec$[ebp+8]        ;
    pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov    DWORD PTR fs:0, eax
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; entering outer try block,
    setting previous try level=0
    mov    DWORD PTR __$SEHRec$[ebp+20], 1 ; entering inner try block,
    setting previous try level=1
    push   OFFSET $SG85497 ; 'hello!'
    call   _printf
    add    esp, 4
    push   0
    push   0
    push   0
    push   1122867 ; 00112233H
    call   DWORD PTR __imp__RaiseException@16
    push   OFFSET $SG85499 ; '0x112233 raised. now let''s crash'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, set
    previous try level back to 0
    jmp    SHORT $LN2@main

; inner block filter:
$LN12@main:
$LN18@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T3[ebp], eax
    cmp    DWORD PTR $T3[ebp], -1073741819 ; c0000005H
    jne    SHORT $LN5@main
```

```
    mov    DWORD PTR tv72[ebp], 1
    jmp    SHORT $LN6@main
$LN5@main:
    mov    DWORD PTR tv72[ebp], 0
$LN6@main:
    mov    eax, DWORD PTR tv72[ebp]
$LN14@main:
$LN16@main:
    ret    0

; inner block handler:
$LN13@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85501 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, setting
    previous try level back to 0
$LN2@main:
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting
    previous try level back to -2
    jmp    SHORT $LN7@main

; outer block filter:
$LN8@main:
$LN19@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T2[ebp], eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    push   ecx
    mov    edx, DWORD PTR $T2[ebp]
    push   edx
    call   _filter_user_exceptions
    add    esp, 8
$LN10@main:
$LN17@main:
    ret    0

; outer block handler:
$LN9@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85503 ; 'user exception caught'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting
    previous try level back to -2
$LN7@main:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:0, ecx
    pop    ecx
    pop    edi
```

```
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP

_code$ = 8  ; size = 4
_ep$ = 12   ; size = 4
_filter_user_exceptions PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _code$[ebp]
    push   eax
    push   OFFSET $SG85486 ; 'in filter. code=0x%08X'
    call   _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867 ; 00112233H
    jne    SHORT $LN2@filter_use
    push   OFFSET $SG85488 ; 'yes, that is our exception'
    call   _printf
    add    esp, 4
    mov    eax, 1
    jmp    SHORT $LN3@filter_use
    jmp    SHORT $LN3@filter_use
$LN2@filter_use:
    push   OFFSET $SG85490 ; 'not our exception'
    call   _printf
    add    esp, 4
    xor    eax, eax
$LN3@filter_use:
    pop    ebp
    ret    0
_filter_user_exceptions ENDP
```

La signification de *cookies* est la suivante: `Cookie Offset` est la différence entre l'adresse dans la pile de la dernière valeur sauvegardée du registre EBP et de l'adresse dans la pile du résultat de l'addition $EBP \oplus security_cookie$. `Cookie XOR Offset` est quant à lui la différence entre $EBP \oplus security_cookie$ et la valeur conservée sur la pile.

Si le prédicat ci-dessous n'est pas respecté, le processus est arrêté du fait d'une corruption de la pile:

$$security_cookie \oplus (CookieXOROffset + address_of_saved_EBP) == \\ stack[address_of_saved_EBP + CookieOffset]$$

Lorsque `Cookie Offset` vaut -2, ceci indique qu'il n'est pas présent.

Windows x64

Vous imaginez bien qu'il n'est pas très performant de construire le contexte SEH dans le prologue de chaque fonction. S'y ajoute les nombreux changements de la valeur de *previous try level* durant l'exécution de la fonction.

C'est pourquoi avec x64, la manière de faire a complètement changé. Tous les pointeurs vers les blocs `try`, les filtres et les gestionnaires sont désormais stockés dans une nouveau segment PE: `.pdata` à partir duquel les gestionnaires d'exception de l'OS récupéreront les informations.

Voici deux exemples tirés de la section précédente et compilés pour x64:

Listing 6.33 : MSVC 2012

```
$SG86276 DB     'hello #1!', 0aH, 00H
$SG86277 DB     'hello #2!', 0aH, 00H
$SG86279 DB     'access violation, can''t recover', 0aH, 00H

pdata   SEGMENT
$pdata$main DD  imagerel $LN9
        DD      imagerel $LN9+61
        DD      imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
        DD      imagerel main$filt$0+32
        DD      imagerel $unwind$main$filt$0
pdata   ENDS
xdata   SEGMENT
$unwind$main DD 020609H
        DD      030023206H
        DD      imagerel __C_specific_handler
        DD      01H
        DD      imagerel $LN9+8
        DD      imagerel $LN9+40
        DD      imagerel main$filt$0
        DD      imagerel $LN9+40
$unwind$main$filt$0 DD 020601H
        DD      050023206H
xdata   ENDS

_TEXT   SEGMENT
main    PROC
$LN9:
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea     rcx, OFFSET FLAT:$SG86276 ; 'hello #1!'
        call    printf
        mov     DWORD PTR [rbx], 13
        lea     rcx, OFFSET FLAT:$SG86277 ; 'hello #2!'
        call    printf
        jmp     SHORT $LN8@main
$LN6@main:
        lea     rcx, OFFSET FLAT:$SG86279 ; 'access violation, can''t
    recover'
        call    printf
        npad    1 ; align next label
$LN8@main:
        xor     eax, eax
```

```
        add     rsp, 32
        pop     rbx
        ret     0
main    ENDP
_TEXT   ENDS

text$x  SEGMENT
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN5@main$filt$:
        mov     rax, QWORD PTR [rcx]
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H
        sete    cl
        mov     eax, ecx
$LN7@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$0 ENDP
text$x  ENDS
```

Listing 6.34 : MSVC 2012

```
$SG86277 DB     'in filter. code=0x%08X', 0aH, 00H
$SG86279 DB     'yes, that is our exception', 0aH, 00H
$SG86281 DB     'not our exception', 0aH, 00H
$SG86288 DB     'hello!', 0aH, 00H
$SG86290 DB     '0x112233 raised. now let''s crash', 0aH, 00H
$SG86292 DB     'access violation, can''t recover', 0aH, 00H
$SG86294 DB     'user exception caught', 0aH, 00H

pdata   SEGMENT
$pdata$filter_user_exceptions DD imagerel $LN6
        DD      imagerel $LN6+73
        DD      imagerel $unwind$filter_user_exceptions
$pdata$main DD  imagerel $LN14
        DD      imagerel $LN14+95
        DD      imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
        DD      imagerel main$filt$0+32
        DD      imagerel $unwind$main$filt$0
$pdata$main$filt$1 DD imagerel main$filt$1
        DD      imagerel main$filt$1+30
        DD      imagerel $unwind$main$filt$1
pdata   ENDS

xdata   SEGMENT
$unwind$filter_user_exceptions DD 020601H
```

```
        DD      030023206H
$unwind$main DD 020609H
        DD      030023206H
        DD      imagerel __C_specific_handler
        DD      02H
        DD      imagerel $LN14+8
        DD      imagerel $LN14+59
        DD      imagerel main$filt$0
        DD      imagerel $LN14+59
        DD      imagerel $LN14+8
        DD      imagerel $LN14+74
        DD      imagerel main$filt$1
        DD      imagerel $LN14+74
$unwind$main$filt$0 DD 020601H
        DD      050023206H
$unwind$main$filt$1 DD 020601H
        DD      050023206H
xdata   ENDS

_TEXT   SEGMENT
main    PROC
$LN14:
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea     rcx, OFFSET FLAT:$SG86288 ; 'hello!'
        call    printf
        xor     r9d, r9d
        xor     r8d, r8d
        xor     edx, edx
        mov     ecx, 1122867 ; 00112233H
        call    QWORD PTR __imp_RaiseException
        lea     rcx, OFFSET FLAT:$SG86290 ; '0x112233 raised. now let''s
    crash'
        call    printf
        mov     DWORD PTR [rbx], 13
        jmp     SHORT $LN13@main
$LN11@main:
        lea     rcx, OFFSET FLAT:$SG86292 ; 'access violation, can''t
    recover'
        call    printf
        npad    1 ; align next label
$LN13@main:
        jmp     SHORT $LN9@main
$LN7@main:
        lea     rcx, OFFSET FLAT:$SG86294 ; 'user exception caught'
        call    printf
        npad    1 ; align next label
$LN9@main:
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
main    ENDP
```

```
text$x  SEGMENT
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN10@main$filt$:
        mov     rax, QWORD PTR [rcx]
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H
        sete    cl
        mov     eax, ecx
$LN12@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$0 ENDP

main$filt$1 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN6@main$filt$:
        mov     rax, QWORD PTR [rcx]
        mov     rdx, rcx
        mov     ecx, DWORD PTR [rax]
        call    filter_user_exceptions
        npad    1 ; align next label
$LN8@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$1 ENDP
text$x  ENDS

_TEXT   SEGMENT
code$ = 48
ep$ = 56
filter_user_exceptions PROC
$LN6:
        push    rbx
        sub     rsp, 32
        mov     ebx, ecx
        mov     edx, ecx
        lea     rcx, OFFSET FLAT:$SG86277 ; 'in filter. code=0x%08X'
        call    printf
        cmp     ebx, 1122867; 00112233H
        jne     SHORT $LN2@filter_use
        lea     rcx, OFFSET FLAT:$SG86279 ; 'yes, that is our exception'
        call    printf
        mov     eax, 1
```

```
        add     rsp, 32
        pop     rbx
        ret     0
$LN2@filter_use:
        lea     rcx, OFFSET FLAT:$SG86281 ; 'not our exception'
        call    printf
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
filter_user_exceptions ENDP
_TEXT   ENDS
```

Pour plus d'informations sur le sujet, lisez [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)] [50].

Hormis les informations d'exception, .pdata est aussi une section qui contient les adresses de début et de fin de toutes les fonctions. Elle revêt donc un intérêt particulier dans le cadre d'une analyse automatique d'un programme.

En lire plus sur SEH

[Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)][51], [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)] [52].

6.5.4 Windows NT: Section critique

Dans tout OS, les sections critiques sont très importantes dans un système multi-threadé, principalement pour donner la garantie qu'un seul thread peut accéder à certaines données à un instant précis, en bloquant les autres threads et les interruptions.

Voilà comment une structure CRITICAL_SECTION est déclarée dans la série des OS Windows NT :

Listing 6.35 : (Windows Research Kernel v1.2) public/sdk/inc/nturtl.h

```
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;

    //
    // The following three fields control entering and exiting the critical
    // section for the resource
    //

    LONG LockCount;
```

[50] Aussi disponible en http://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf
[51] Aussi disponible en http://www.microsoft.com/msj/0197/Exception/Exception.aspx
[52] Aussi disponible en http://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf

```
    LONG RecursionCount;
    HANDLE OwningThread;        // from the thread's ClientId->UniqueThread
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount;        // force size on 64-bit systems when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

Voilà comment la fonction EnterCriticalSection() fonctionne:

Listing 6.36 : Windows 2008/ntdll.dll/x86 (begin)

```
_RtlEnterCriticalSection@4

var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
arg_0           = dword ptr  8

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                sub     esp, 0Ch
                push    esi
                push    edi
                mov     edi, [ebp+arg_0]
                lea     esi, [edi+4] ; LockCount
                mov     eax, esi
                lock btr dword ptr [eax], 0
                jnb     wait ; jump if CF=0

loc_7DE922DD:
                mov     eax, large fs:18h
                mov     ecx, [eax+24h]
                mov     [edi+0Ch], ecx
                mov     dword ptr [edi+8], 1
                pop     edi
                xor     eax, eax
                pop     esi
                mov     esp, ebp
                pop     ebp
                retn    4

... skipped
```

L'instruction la plus importante dans ce morceau de code est BTR (préfixée avec LOCK) :

Le bit d'index zéro est stocké dans le flag CF et est effacé en mémoire. Ceci est une opération atomique, bloquant tous les autres accès du CPU à cette zone de mémoire (regardez le préfixe LOCK se trouvant avant l'instruction BTR). Si le bit en LockCount est 1, bien, remise à zéro et retour de la fonction: nous sommes dans une section critique.

Si non—la section critique est déjà occupée par un autre thread, donc attendre. L'attente est effectuée en utilisant WaitForSingleObject().

Et voici comment la fonction LeaveCriticalSection() fonctionne:

Listing 6.37 : Windows 2008/ntdll.dll/x86 (begin)

```
_RtlLeaveCriticalSection@4 proc near

arg_0           = dword ptr  8

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    esi
                mov     esi, [ebp+arg_0]
                add     dword ptr [esi+8], 0FFFFFFFFh ; RecursionCount
                jnz     short loc_7DE922B2
                push    ebx
                push    edi
                lea     edi, [esi+4]    ; LockCount
                mov     dword ptr [esi+0Ch], 0
                mov     ebx, 1
                mov     eax, edi
                lock xadd [eax], ebx
                inc     ebx
                cmp     ebx, 0FFFFFFFFh
                jnz     loc_7DEA8EB7

loc_7DE922B0:
                pop     edi
                pop     ebx

loc_7DE922B2:
                xor     eax, eax
                pop     esi
                pop     ebp
                retn    4

... skipped
```

XADD signifie «exchange and add » (échanger et ajouter).

Dans ce cas, elle ajoute 1 à LockCount, en même temps qu'elle sauve la valeur initiale de LockCount dans le registre EBX. Toutefois, la valeur dans EBX est incrémentée avec l'aide du INC EBX subséquent, et il sera ainsi égal à la valeur modifiée de LockCount.

Cette opération est atomique puisqu'elle est préfixée par LOCK, signifiant que tous les autres CPUs ou cœurs de CPU dans le système ne peuvent pas accéder à cette zone de la mémoire.

Le préfixe LOCK est très important:

sans lui deux threads, travaillant chacune sur un CPU ou un cœur de CPU séparé pourraient essayer d'entrer dans la section critique et de modifier la valeur en mémoire, ce qui résulterait en un comportement non déterministe.

Chapitre 7

Outils

> Maintenant que Dennis Yurichev a réalisé ce livre gratuit, il s'agit d'une contribution au monde de la connaissance et de l'éducation gratuite. Cependant, pour l'amour de la liberté, nous avons besoin d'outils de rétro-ingénierie (libres) afin de remplacer les outils propriétaires mentionnés dans ce livre.
>
> Richard M. Stallman

7.1 Analyse statique

Outils à utiliser lorsqu'aucun processus n'est en cours d'exécution.

- (Gratuit, open-source) *ent*[1] : outil d'analyse d'entropie. En savoir plus sur l'entropie : 9.2 on page 1225.
- *Hiew*[2] : pour de petites modifications de code dans les fichiers binaires. Inclut un assembleur/désassembleur.
- Libre, open-source *GHex*[3] : éditeur hexadécimal simple pour Linux.
- (Libre, open-source) *xxd* et *od* : utilitaires standards UNIX pour réaliser un dump.
- (Libre, open-source) *strings* : outil *NIX pour rechercher des chaînes ASCII dans des fichiers binaires, fichiers exécutables inclus. Sysinternals ont une alternative [4] qui supporte les larges chaînes de caractères (UTF-16, très utilisé dans Windows).
- (Libre, open-source) *Binwalk*[5] : analyser les images firmware.

[1] http://www.fourmilab.ch/random/
[2] hiew.ru
[3] https://wiki.gnome.org/Apps/Ghex
[4] https://technet.microsoft.com/en-us/sysinternals/strings
[5] http://binwalk.org/

- (Libre, open-source) *binary grep* : un petit utilitaire pour rechercher une séquence d'octets dans un paquet de fichiers, incluant ceux non exécutables : GitHub. Il y a aussi rafind2 dans rada.re pour le même usage.

7.1.1 Désassembleurs

- *IDA*. Une ancienne version Freeware est disponible via téléchargement [6]. Antisèche des touches de raccourci: .6.1 on page 1354
- (Gratuit, open-source) *Ghidra*[7] — une alternative libre et open-source de IDA développée par la NSA[8].
- *Binary Ninja*[9]
- (Gratuit, open-source) *zynamics BinNavi*[10]
- (Gratuit, open-source) *objdump* : simple utilitaire en ligne de commandes pour désassembler et réaliser des dumps.
- (Gratuit, open-source) *readelf*[11] : réaliser des dumps d'informations sur des fichiers ELF.

7.1.2 Décompilateurs

Il n'existe qu'un seul décompilateur connu en C, d'excellente qualité et disponible au public *Hex-Rays* :
hex-rays.com/products/decompiler/

Pour en savoir plus: 11.9 on page 1301.

Il y a une alternative libre développée par la NSA : *Ghidra*[12].

7.1.3 Comparaison de versions

Vous pouvez éventuellement les utiliser lorsque vous comparez la version originale d'un exécutable et une version remaniée, pour déterminer ce qui a été corrigé et en déterminer la raison.

- (Gratuit) *zynamics BinDiff*[13]
- (Gratuit, open-source) *Diaphora*[14]

[6] hex-rays.com/products/ida/support/download_Freeware.shtml
[7] https://ghidra-sre.org/
[8] National Security Agency (Agence Nationale de la Sécurité)
[9] http://binary.ninja/
[10] https://www.zynamics.com/binnavi.html
[11] https://sourceware.org/binutils/docs/binutils/readelf.html
[12] https://ghidra-sre.org/
[13] https://www.zynamics.com/software.html
[14] https://github.com/joxeankoret/diaphora

7.2 Analyse dynamique

Outils à utiliser lorsque que le système est en cours d'exploitation ou lorsqu'un processus est en cours d'exécution.

7.2.1 Débogueurs

- (Gratuit) *OllyDbg*. Débogueur Win32 très populaire [15]. Anti-sèche des touches de raccourci: .6.2 on page 1355
- (Gratuit, open-source) *GDB*. Débogueur peu populaire parmi les ingénieurs en rétro-ingénierie, car il est principalement destiné aux programmeurs. Quelques commandes : .6.5 on page 1356. Il y a une interface graphique pour GDB, "GDB dashboard"[16].
- (Gratuit, open-source) *LLDB*[17].
- *WinDbg*[18] : débogueur pour le noyau Windows.
- (Gratuit, open-source) *Radare* AKA rada.re AKA r2[19]. Une interface graphique existe aussi : *ragui*[20].
- (Gratuit, open-source) *tracer*. L'auteur utilise souvent *tracer* [21] au lieu d'un débogueur.

  L'auteur de ces lignes a finalement arrêté d'utiliser un débogueur, depuis que tout ce dont il a besoin est de repérer les arguments d'une fonction lorsque cette dernière est exécutée, ou l'état des registres à un instant donné. Le temps de chargement d'un débogueur étant trop long, un petit utilitaire sous le nom de *tracer* a été conçu. Il fonctionne depuis la ligne de commandes, permettant d'intercepter l'exécution d'une fonction, en plaçant des breakpoints à des endroits définis, en lisant et en changeant l'état des registres, etc...

  N.B.: *tracer* n'évolue pas, parce qu'il a été développé en tant qu'outil de démonstration pour ce livre, et non pas comme un outil dont on se servirait au quotidien.

7.2.2 Tracer les appels de librairies

ltrace[22].

[15] ollydbg.de
[16] https://github.com/cyrus-and/gdb-dashboard
[17] http://lldb.llvm.org/
[18] https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit
[19] http://rada.re/r/
[20] http://radare.org/ragui/
[21] yurichev.com
[22] http://www.ltrace.org/

7.2.3 Tracer les appels système

strace / dtruss

Montre les appels système (syscalls(6.3 on page 972)) effectués dans l'immédiat.

Par exemple:

```
# strace df -h

...

access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or ↙
    ↳ directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF↙
    ↳ \1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232\1\0004\0\0\0"..., ↙
    ↳ 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) ↙
    ↳ = 0xb75b3000
```

Mac OS X a dtruss pour faire la même chose.

Cygwin a également strace, mais de ce que je sais, cela ne fonctionne que pour les fichiers .exe compilés pour l'environnement Cygwin lui-même.

7.2.4 Sniffer le réseau

Sniffer signifie intercepter des informations qui peuvent vous intéresser.

(Gratuit, open-source) *Wireshark*[23] pour sniffer le réseau. Peut également sniffer les protocoles USB [24].

Wireshark a un petit (ou vieux) frère *tcpdump*[25], outil simple en ligne de commandes.

7.2.5 Sysinternals

(Gratuit) Sysinternals (développé par Mark Russinovich) [26]. Ces outils sont importants et valent la peine d'être étudiés : Process Explorer, Handle, VMMap, TCPView, Process Monitor.

7.2.6 Valgrind

(Gratuit, open-source) un puissant outil pour détecter les fuites mémoire : http://valgrind.org/. Grâce à ses puissants mécanismes JIT ("Just In Time"), Valgrind est utilisé comme un framework pour d'autres outils.

23 https://www.wireshark.org/
24 https://wiki.wireshark.org/CaptureSetup/USB
25 http://www.tcpdump.org/
26 https://technet.microsoft.com/en-us/sysinternals/bb842062

7.2.7 Emulateurs

- (Gratuit, open-source) *QEMU*[27] : émulateur pour différents CPUs et architectures.
- (Gratuit, open-source) *DosBox*[28] : émulateur MS-DOS, principalement utilisé pour le rétro-gaming.
- (Gratuit, open-source) *SimH*[29] : émulateur d'anciens ordinateurs, unités centrales, etc...

7.3 Autres outils

Microsoft Visual Studio Express [30] : Version gratuite simplifiée de Visual Studio, pratique pour des études de cas simples.

Quelques options utiles : .6.3 on page 1355.

Il y a un site web appelé "Compiler Explorer", permettant de compiler des petits morceaux de code et de voir le résultat avec des versions variées de GCC et d'architectures (au moins x86, ARM, MIPS) : `http://godbolt.org/`—Je l'aurais utilisé pour le livre si je l'avais connu!

7.3.1 Solveurs SMT

Du point de vue de rétro-ingénieur, les solveurs SAT sont utilisés lorsque l'on fait face à de la cryptogrphie amateur, de l'exécution symoblique/concolique, de la génération de chaîne ROP.

Pour plus d'information, lire: `https://yurichev.com/writings/SAT_SMT_by_example.pdf`.

7.3.2 Calculatrices

Une bonne calculatrice pour les besoins des rétro-ingénieurs doit au moins supporter les bases décimale, hexadécimale et binaire, ainsi que plusieurs opérations importantes comme XOR et les décalages.

- IDA possède une calculatrice intégrée ("?").
- rada.re a *rax2*.
- `https://yurichev.com/progcalc/`
- En dernier recours, la calculatrice standard de Windows dispose d'un mode programmeur.

[27] `http://qemu.org`
[28] `https://www.dosbox.com/`
[29] `http://simh.trailing-edge.com/`
[30] visualstudio.com/en-US/products/visual-studio-express-vs

7.4 Un outil manquant ?

Si vous connaissez un bon outil non listé précédemment, n’hésitez pas à m’en faire la remarque :
my emails.

Chapitre 8

Études de cas

Plutôt qu'un épigraphe:

> **Peter Seibel:** Comment vous attaquez-vous à la lecture de code source? Même lire quelque chose dans un langage de programmation que vous connaissez déjà est un problème délicat.
>
> **Donald Knuth:** Mais ça vaut vraiment la peine pour ce que ça construit dans votre cerveau. Donc, comment est-ce que je fais? Il y avait une machine appelée le Bunker Ramo 300 et quelqu'un m'avait dit que le compilateur ForTran pour cette machine était incroyablement rapide, mais personne n'avait la moindre idée de pourquoi il fonctionnait. Je me suis procuré une copie du listing de son code source. Je n'avais pas de manuel pour la machine, donc je n'étais même pas sûr de ce qu'était son langage machine.
>
> Mais j'ai pris ça comme un défi intéressant. J'ai pu découvrir BEGIN et j'ai alors commencé à décoder. Les codes opération avaient des sortes de mnémoniques sur deux lettres et donc j'ai pu commencer à comprendre que "Ceci était probablement une instruction de chargement, ceci probablement un branchement". Et je savais qu'il s'agissait d'un compilateur ForTran, donc à un moment donné j'ai regardé la colonne sept d'une carte, et c'était où ça disait s'il s'agissait d'un commentaire ou non.
>
> Après trois heures, j'en avais découvert un peu à propos de cette machine. Alors, j'ai trouvé cette grosse table de branchement. Donc, c'était un puzzle et j'ai continué à faire des petits graphiques comme si je travaillais dans un un organisme de sécurité essayant de décoder un code secret. Mais je savais que ça fonctionnait et je savais que c'était un compilateur ForTran-ce n'était pas chiffré dans le sens où ça serait volontairement opaque; c'était seulement du code car je n'avais pas reçu le manuel de cette machine.
>
> Enfin j'ai réussi à comprendre pourquoi ce compilateur était si rapide. Malheureusement ce n'était pas parce que son algorithme était

> brillant; c'était seulement parce qu'ils avaient utilisé une programmation non structurée et optimisé le code manuellement.
>
> C'était simplement la façon de résoudre une énigme inconnue-faire des tableaux et des graphiques et y obtenir un peu plus d'informations et faire une hypothèse. En général lorsque je lis un papier technique, c'est le même défi. J'essaye de me mettre dans l'esprit de l'auteur, pour essayer de comprendre ce qu'est le concept. Plus vous apprenez à lire les trucs des autres, plus vous serez capable d'inventer les votre dans le futur, il me semble.

(Peter Seibel — Coders at Work: Reflections on the Craft of Programming)[1]

8.1 Blague avec le solitaire Mahjong (Windows 7)

Le solitaire Mahjong est un bon jeu, mais pouvons-nous le rendre plus difficile, en désactivant l'élément de menu *Hint* ?

Dans mon Windows 7, je peux trouver Mahjong.dll et Mahjong.exe dans:
`C:\Windows\winsxs\`
`x86_microsoft-windows-s..inboxgames-shanghai_31bf3856ad364e35_6.1.7600.16385_none\`
`c07a51d9507d9398`.

Aussi le fichier `Mahjong.exe.mui` dans:
`C:\Windows\winsxs\`
`x86_microsoft-windows-s..-shanghai.resources_31bf3856ad364e35_6.1.7600.16385_en-us`
`_c430954533c66bf3`

et

`C:\Windows\winsxs\`
`x86_microsoft-windows-s..-shanghai.resources_31bf3856ad364e35_6.1.7600.16385_ru-ru`
`_0d51acf984cb679a`.

J'utilise Windows en anglais, mais avec le support du langage russe, donc il peut y avoir des fichiers de ressource pour ces deux langages. En ouvrant Mahjong.exe.mui dans Resource Hacker, nous pouvons y voir une définition de menu:

Listing 8.1 : Ressource de menu de Mahjong.exe.mui

```
103 MENU
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
{
  POPUP "&Game"
  {
    MENUITEM "&New Game\tF2",  40000
    MENUITEM SEPARATOR
    MENUITEM "&Undo\tCtrl+Z",  40001
    MENUITEM "&Hint\tH",  40002
    MENUITEM SEPARATOR
```

[1] NDT: ouvrage non traduit en français, la traduction, et les fautes, sont miennes.

```
    MENUITEM "&Statistics\tF4",  40003
    MENUITEM "&Options\tF5",  40004
    MENUITEM "Change &Appearance\tF7",  40005
    MENUITEM SEPARATOR
    MENUITEM "E&xit",  40006
  }
  POPUP "&Help"
  {
    MENUITEM "&View Help\tF1",  40015
    MENUITEM "&About Mahjong Titans",  40016
    MENUITEM SEPARATOR
    MENUITEM "Get &More Games Online",  40020
  }
}
```

Le sous-menu *Hint* a le code 40002. Maintenant, j'ouvre Mahjong.exe dans IDA et trouve la valeur 40002.

(J'écris ceci en novembre 2019. Il semble qu'IDA ne puisse pas obtenir les PDBs depuis les serveurs de Microsoft. Peut-être que Windows 7 n'est plus supporté? En tout cas, je ne peux pas obtenir les noms de fonction...)

Listing 8.2 : Mahjong.exe

```
.text:010205C8 6A 03                push    3
.text:010205CA 85 FF                test    edi, edi
.text:010205CC 5B                   pop     ebx

...

.text:01020625 57                   push    edi     ; uIDEnableItem
.text:01020626 FF 35 C8 97 08 01    push    hmenu   ; hMenu
.text:0102062C FF D6                call    esi     ; EnableMenuItem
.text:0102062E 83 7D 08 01          cmp     [ebp+arg_0], 1
.text:01020632 BF 42 9C 00 00       mov     edi, 40002
.text:01020637 75 18                jnz     short loc_1020651 ; must jump
    always
.text:01020639 6A 00                push    0       ; uEnable
.text:0102063B 57                   push    edi     ; uIDEnableItem
.text:0102063C FF 35 B4 8B 08 01    push    hMenu   ; hMenu
.text:01020642 FF D6                call    esi     ; EnableMenuItem
.text:01020644 6A 00                push    0       ; uEnable
.text:01020646 57                   push    edi     ; uIDEnableItem
.text:01020647 FF 35 C8 97 08 01    push    hmenu   ; hMenu
.text:0102064D FF D6                call    esi     ; EnableMenuItem
.text:0102064F EB 1A                jmp     short loc_102066B
.text:01020651
.text:01020651                     loc_1020651:     ; CODE XREF: sub_1020581+B6
.text:01020651 53                   push    ebx     ; 3
.text:01020652 57                   push    edi     ; uIDEnableItem
.text:01020653 FF 35 B4 8B 08 01    push    hMenu   ; hMenu
.text:01020659 FF D6                call    esi     ; EnableMenuItem
.text:0102065B 53                   push    ebx     ; 3
.text:0102065C 57                   push    edi     ; uIDEnableItem
```

```
.text:0102065D FF 35 C8 97 08 01  push    hmenu  ; hMenu
.text:01020663 FF D6              call    esi    ; EnableMenuItem
```

Ce morceau de code active ou désactive l'élément de menu *Hint*.

Et d'après MSDN[2] :

`MF_DISABLED | MF_GRAYED = 3` et `MF_ENABLED = 0`.

Je pense que cette fonction active ou désactive plusieurs élément de menu (*Hint*, *Undo*, etc), suivant la valeur dans `arg_0`. Car au début, lorsqu'un utilisateur choisi le type de solitaire `Hint` et `Undo` sont désactivés. Ils sont activés lorsque le jeu a commencé.

Je modifie le fichier Mahjong.exe en 0x01020637, en remplaçant l'octet 0x75 avec 0xEB, rendant ce saut `JNZ` toujours pris. Pratiquement, ceci va toujours appeler `EnableMenuItem(..., ..., 3)`. Maintenant le sous-menu *Hint* est toujours désactivé.

Aussi, de manière ou d'une autre, `EnableMenuItem()` est appelé deux fois, pour `hMenu` et pour `hmenu`. Peut-être que le programme a deux menus, et peut-être les échange-t-il?

À titre d'exercice, essayez de désactiver l'élément de menu *Undo*, pour rendre le jeu encore plus difficile.

8.2 Blague avec le gestionnaire de tâche (Windows Vista)

Voyons s'il est possible de légèrement modifier le gestionnaire de tâches pour qu'il détecte plus de cœurs CPU.

Demandons-nous d'abord, comment est-ce que le gestionnaire de tâches connait le nombre de cœurs?

Il y a une fonction win32 `GetSystemInfo()` présente dans l'espace utilisateur win32 qui peut nous dire ceci. Mais elle n'est pas importées dans `taskmgr.exe`.

Il y en a, toutefois, une autre dans NTAPI, `NtQuerySystemInformation()`, qui est utilisée dans `taskmgr.exe` à plusieurs endroits.

Pour obtenir le nombre de cœurs, il faut appeler cette fonction avec la constante `SystemBasicInformation` comme premier argument (qui vaut zéro[3]).

Le second argument doit pointer vers le buffer qui va recevoir toute l'information.

Donc nous devons trouver tous les appels à la fonction `NtQuerySystemInformation(0, ?, ?, ?)`. Ouvrons `taskmgr.exe` dans IDA.

Ce qui est toujours bien avec les exécutables Microsoft, c'est que IDA peut télécharger le fichier PDB correspondant à cet exécutable et afficher les noms de toutes les fonctions.

[2] https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-enablemenuitem
[3] MSDN

Il est visible que le gestionnaire des tâches est écrit en C++ et certains noms de fonction et classes sont vraiment parlants. Il y a des classes CAdapter, CNetPage, CPerfPage, CProcInfo, CProcPage, CSvcPage, CTaskPage, CUserPage.

Il semble que chaque onglet du gestionnaire de tâches ait une classe correspondante.

Regardons chaque appel et ajoutons un commentaire avec la valeur qui est passée comme premier argument de la fonction. Nous allons écrire «not zero» à certains endroits, car la valeur n'est clairement pas zéro, mais quelque chose de vraiment différent (plus à ce propos dans la seconde partie de ce chapitre).

Et nous cherchons les zéros passés comme argument après tout.

xrefs to __imp_NtQuerySystemInformation

Dire...	T.	Address	Text
Up	p	wWinMain+50E	call cs:__imp_NtQuerySystemInformation; 0
Up	p	wWinMain+542	call cs:__imp_NtQuerySystemInformation; 2
Up	p	CPerfPage::TimerEvent(void)+200	call cs:__imp_NtQuerySystemInformation; not zero
	p	InitPerfInfo(void)+2C	call cs:__imp_NtQuerySystemInformation; 0
D...	p	InitPerfInfo(void)+F0	call cs:__imp_NtQuerySystemInformation; 8
D...	p	CalcCpuTime(int)+5F	call cs:__imp_NtQuerySystemInformation; 8
D...	p	CalcCpuTime(int)+248	call cs:__imp_NtQuerySystemInformation; 2
D...	p	CPerfPage::CalcPhysicalMem(unsigned ...	call cs:__imp_NtQuerySystemInformation; not zero
D...	p	CPerfPage::CalcPhysicalMem(unsigned ...	call cs:__imp_NtQuerySystemInformation; not zero
D...	p	CProcPage::GetProcessInfo(void)+2B	call cs:__imp_NtQuerySystemInformation; 5
D...	p	CProcPage::UpdateProcInfoArray(void)+...	call cs:__imp_NtQuerySystemInformation; 0
D...	p	CProcPage::UpdateProcInfoArray(void)+...	call cs:__imp_NtQuerySystemInformation; 2
D...	p	CProcPage::Initialize(HWND__ *)+201	call cs:__imp_NtQuerySystemInformation; 0
D...	p	CProcPage::GetTaskListEx(void)+3C	call cs:__imp_NtQuerySystemInformation; 5

Fig. 8.1: IDA: références croisées vers NtQuerySystemInformation()

Oui, les noms parlent vraiment d'eux-même.

Nous allons examiner précisément les endroits où `NtQuerySystemInformation(0, ?, ?, ?)` est appelée, nous trouvons rapidement ce que nous cherchons dans la fonction `InitPerfInfo()` :

Listing 8.3 : taskmgr.exe (Windows Vista)

```
.text:10000B4B3          xor     r9d, r9d
.text:10000B4B6          lea     rdx, [rsp+0C78h+var_C58] ; buffer
.text:10000B4BB          xor     ecx, ecx
.text:10000B4BD          lea     ebp, [r9+40h]
.text:10000B4C1          mov     r8d, ebp
.text:10000B4C4          call    cs:__imp_NtQuerySystemInformation ; 0
.text:10000B4CA          xor     ebx, ebx
.text:10000B4CC          cmp     eax, ebx
.text:10000B4CE          jge     short loc_10000B4D7
```

```
.text:10000B4D0
.text:10000B4D0 loc_10000B4D0:                          ; CODE XREF:
    InitPerfInfo(void)+97
.text:10000B4D0                                         ;
    InitPerfInfo(void)+AF
.text:10000B4D0          xor     al, al
.text:10000B4D2          jmp     loc_10000B5EA
.text:10000B4D7 ;
    ---------------------------------------------------------------------------
.text:10000B4D7
.text:10000B4D7 loc_10000B4D7:                          ; CODE XREF:
    InitPerfInfo(void)+36
.text:10000B4D7          mov     eax, [rsp+0C78h+var_C50]
.text:10000B4DB          mov     esi, ebx
.text:10000B4DD          mov     r12d, 3E80h
.text:10000B4E3          mov     cs:?g_PageSize@@3KA, eax ; ulong g_PageSize
.text:10000B4E9          shr     eax, 0Ah
.text:10000B4EC          lea     r13, __ImageBase
.text:10000B4F3          imul    eax, [rsp+0C78h+var_C4C]
.text:10000B4F8          cmp     [rsp+0C78h+var_C20], bpl
.text:10000B4FD          mov     cs:?g_MEMMax@@3_JA, rax ; __int64 g_MEMMax
.text:10000B504          movzx   eax, [rsp+0C78h+var_C20] ; number of CPUs
.text:10000B509          cmova   eax, ebp
.text:10000B50C          cmp     al, bl
.text:10000B50E          mov     cs:?g_cProcessors@@3EA, al ;
    uchar g_cProcessors
```

g_cProcessors est une variable globale, et ce nom a été assigné par IDA suivant le symbole PDB chargé depuis le serveur de Microsoft.

L'octet est pris de var_C20. Et var_C58 est passée à NtQuerySystemInformation() comme un pointeur sur le buffer de réception. La différence entre 0xC20 et 0xC58 est 0x38 (56).

Regardons le format de la structure renvoyée, que nous pouvons trouver dans MSDN:

```
typedef struct _SYSTEM_BASIC_INFORMATION {
    BYTE Reserved1[24];
    PVOID Reserved2[4];
    CCHAR NumberOfProcessors;
} SYSTEM_BASIC_INFORMATION;
```

Ceci est un système x64, donc chaque PVOID occupe 8 octets.

Tous les champs réservés dans la structure occupent $24 + 4 * 8 = 56$ octets.

Oh oui, ceci implique que var_C20 dans la pile locale est exactement le champ NumberOfProcessors de la structure SYSTEM_BASIC_INFORMATION.

Vérifions notre hypothèse. Copier taskmgr.exe depuis C:\Windows\System32 dans un autre répertoire (ainsi le *Windows Resource Protection* ne va pas essayer de restaurer l'ancienne version du taskmgr.exe modifié).

Ouvrons-le dans Hiew et trouvons l'endroit:

```
01`0000B4F8: 40386C2458                    cmp         [rsp][058],bpl
01`0000B4FD: 48890544A00100                mov         [00000001`00025548],rax
01`0000B504: 0FB6442458                    movzx       eax,b,[rsp][058]
01`0000B509: 0F47C5                        cmova       eax,ebp
01`0000B50C: 3AC3                          cmp         al,bl
01`0000B50E: 880574950100                  mov         [00000001`00024A88],al
01`0000B514: 7645                          jbe        .00000001`0000B55B --▯3
01`0000B516: 488BFB                        mov         rdi,rbx
01`0000B519: 498BD4                       5mov         rdx,r12
01`0000B51C: 8BCD                          mov         ecx,ebp
```

Fig. 8.2: Hiew: trouver l'endroit à modifier

Remplaçons l'instruction MOVZX par la notre. Prétendons avoir un CPU 64 cœurs.

Ajouter un NOP additionnel (car notre instruction est plus courte que l'originale) :

```
00`0000A8F8: 40386C2458                    cmp         [rsp][058],bpl
00`0000A8FD: 48890544A00100                mov         [000024948],rax
00`0000A904: 66B84000                      mov         ax,00040 ;' @'
00`0000A908: 90                            nop
00`0000A909: 0F47C5                        cmova       eax,ebp
00`0000A90C: 3AC3                          cmp         al,bl
00`0000A90E: 880574950100                  mov         [000023E88],al
00`0000A914: 7645                          jbe         00000A95B
00`0000A916: 488BFB                        mov         rdi,rbx
00`0000A919: 498BD4                        mov         rdx,r12
00`0000A91C: 8BCD                          mov         ecx,ebp
```

Fig. 8.3: Hiew: modification effectuée

Et ça fonctionne! Bien sûr, les données dans les graphes ne sont pas correctes.

À certains moments, le gestionnaire de tâches montre même une charge globale du CPU de plus de 100%.

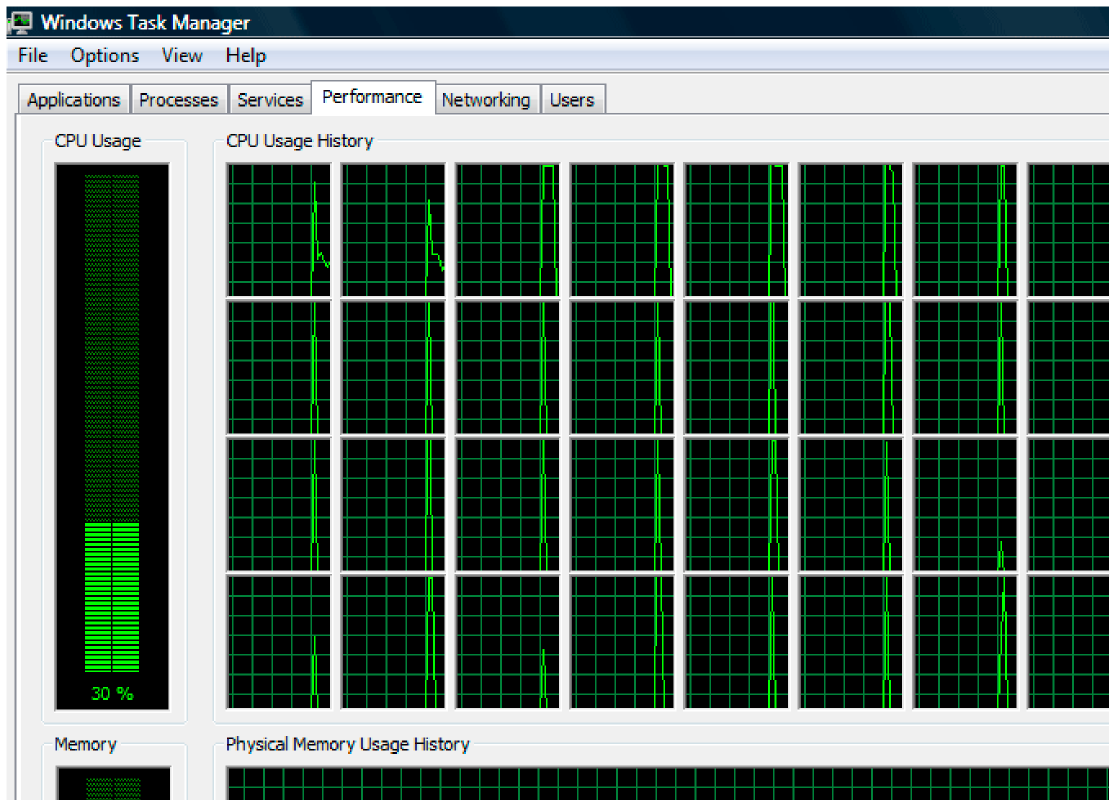


Fig. 8.4: Gestionnaire de tâches Windows fou

Le plus grand nombre avec lequel le gestionnaire de tâches ne plante pas est 64.

Il semble que le gestionnaire de tâche de Windows Vista n'a pas été testé sur des ordinateurs avec un grand nombre de cœurs.

Il doit y avoir une sorte de structure de données dedans. limitée à 64 cœurs (ou plusieurs).

8.2.1 Utilisation de LEA pour charger des valeurs

Parfois, LEA est utilisée dans `taskmgr.exe` au lieu de `MOV` pour définir le premier argument de
`NtQuerySystemInformation()` :

Listing 8.4 : taskmgr.exe (Windows Vista)

```
        xor     r9d, r9d
        div     dword ptr [rsp+4C8h+WndClass.lpfnWndProc]
        lea     rdx, [rsp+4C8h+VersionInformation]
        lea     ecx, [r9+2]      ; put 2 to ECX
        mov     r8d, 138h
        mov     ebx, eax
```

```
; ECX=SystemPerformanceInformation
                call    cs:__imp_NtQuerySystemInformation ; 2

                ...

                mov     r8d, 30h
                lea     r9, [rsp+298h+var_268]
                lea     rdx, [rsp+298h+var_258]
                lea     ecx, [r8-2Dh]   ; put 3 to ECX
; ECX=SystemTimeOfDayInformation
                call    cs:__imp_NtQuerySystemInformation ; not zero

                ...

                mov     rbp, [rsi+8]
                mov     r8d, 20h
                lea     r9, [rsp+98h+arg_0]
                lea     rdx, [rsp+98h+var_78]
                lea     ecx, [r8+2Fh]   ; put 0x4F to ECX
                mov     [rsp+98h+var_60], ebx
                mov     [rsp+98h+var_68], rbp
; ECX=SystemSuperfetchInformation
                call    cs:__imp_NtQuerySystemInformation ; not zero
```

Peut-être que MSVC fit ainsi car le code machine de LEA est plus court que celui de `MOV REG, 5` (il serait de 5 au lieu de 4).

LEA avec un offset dans l'intervalle $-128..127$ (l'offset occupe 1 octet dans l'opcode) avec des registres 32-bit est encore plus court (faute de préfixe REX)—3 octets.

Un autre exemple d'une telle chose: 6.1.5 on page 959.

8.3 Blague avec le jeu Color Lines

Ceci est un jeu très répandu dont il existe plusieurs implémentations. Nous utilisons l'une d'entre elles, appelée BallTriX, de 1997, disponible librement ici https://archive.org/details/BallTriX_1020 [4]. Voici à quoi il ressemble:

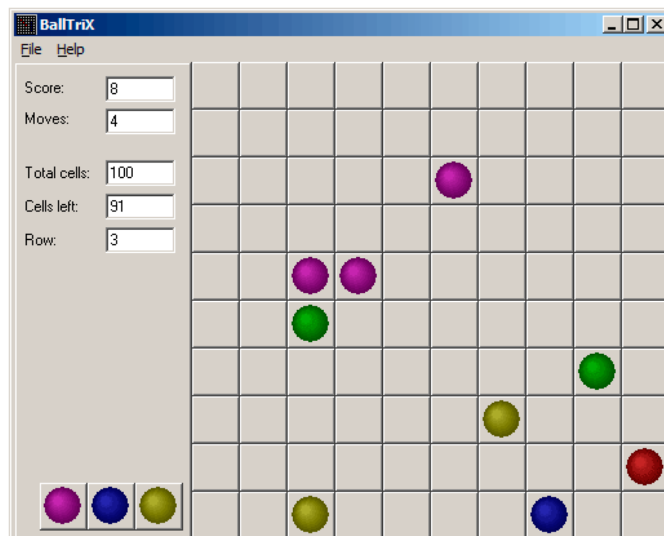


Fig. 8.5: Ceci est l'allure du jeu en général

[4] Ou ici https://web.archive.org/web/20141110053442/http://www.download-central.ws/Win32/Games/B/BallTriX/ ou http://www.benya.com/balltrix/.

Dons regardons s'il est possible de trouver le générateur d'aléas et de jouer des tours avec. IDA reconnaît rapidement la fonction standard _rand dans balltrix.exe en 0x00403DA0. IDA montre aussi qu'elle n'est appelée que d'un seul endroit:

```
.text:00402C9C sub_402C9C      proc near               ;
    CODE XREF: sub_402ACA+52
.text:00402C9C                                         ; sub_402ACA+64 ...
.text:00402C9C
.text:00402C9C arg_0           = dword ptr  8
.text:00402C9C
.text:00402C9C                 push    ebp
.text:00402C9D                 mov     ebp, esp
.text:00402C9F                 push    ebx
.text:00402CA0                 push    esi
.text:00402CA1                 push    edi
.text:00402CA2                 mov     eax, dword_40D430
.text:00402CA7                 imul    eax, dword_40D440
.text:00402CAE                 add     eax, dword_40D5C8
.text:00402CB4                 mov     ecx, 32000
.text:00402CB9                 cdq
.text:00402CBA                 idiv    ecx
.text:00402CBC                 mov     dword_40D440, edx
.text:00402CC2                 call    _rand
.text:00402CC7                 cdq
.text:00402CC8                 idiv    [ebp+arg_0]
.text:00402CCB                 mov     dword_40D430, edx
.text:00402CD1                 mov     eax, dword_40D430
.text:00402CD6                 jmp     $+5
.text:00402CDB                 pop     edi
.text:00402CDC                 pop     esi
.text:00402CDD                 pop     ebx
.text:00402CDE                 leave
.text:00402CDF                 retn
.text:00402CDF sub_402C9C      endp
```

Appelons-la «random». Ne plongeons pas encore dans le code de cette fonction.

Cette fonction est référencée depuis 3 endroits.

Voici les deux premiers:

```
.text:00402B16                 mov     eax, dword_40C03C ; 10 here
.text:00402B1B                 push    eax
.text:00402B1C                 call    random
.text:00402B21                 add     esp, 4
.text:00402B24                 inc     eax
.text:00402B25                 mov     [ebp+var_C], eax
.text:00402B28                 mov     eax, dword_40C040 ; 10 here
.text:00402B2D                 push    eax
.text:00402B2E                 call    random
.text:00402B33                 add     esp, 4
```

Voici le troisième:

```
.text:00402BBB                 mov     eax, dword_40C058 ; 5 here
.text:00402BC0                 push    eax
.text:00402BC1                 call    random
.text:00402BC6                 add     esp, 4
.text:00402BC9                 inc     eax
```

Donc la fonction n'a qu'un argument.

10 est passé dans les deux premiers cas et 5 dans le troisième. Nous pouvons aussi remarquer que le plateau a une taille de 10*10, et qu'il y a 5 couleurs possible. C'est ça! La fonction standard `rand()` renvoie un nombre dans l'intervalle `0..0x7FFF` et c'est souvent peu pratique, donc beaucoup de programmeurs implémentent leur propre fonction qui renvoie un nombre aléatoire dans un intervalle spécifié. Dans notre cas, l'intervalle est $0..n-1$ et n est passé comme unique argument à la fonction. Nous pouvons tester cela rapidement dans le débogueur.

Donc modifions le troisième appel de la fonction, afin qu'il renvoie toujours zéro. Premièrement, nous allons remplacer trois instructions (`PUSH/CALL/ADD`) par des NOPs. Puis, nous allons ajouter l'instruction `XOR EAX, EAX` pour effacer le registre EAX.

```
.00402BB8: 83C410          add         esp,010
.00402BBB: A158C04000      mov         eax,[00040C058]
.00402BC0: 31C0            xor         eax,eax
.00402BC2: 90              nop
.00402BC3: 90              nop
.00402BC4: 90              nop
.00402BC5: 90              nop
.00402BC6: 90              nop
.00402BC7: 90              nop
.00402BC8: 90              nop
.00402BC9: 40              inc         eax
.00402BCA: 8B4DF8          mov         ecx,[ebp][-8]
.00402BCD: 8D0C49          lea         ecx,[ecx][ecx]*2
.00402BD0: 8B15F4D54000    mov         edx,[00040D5F4]
```

Nous avons remplacé l'appel à la fonction `random()` par du code qui renvoie toujours zéro.

Lançons-le maintenant:

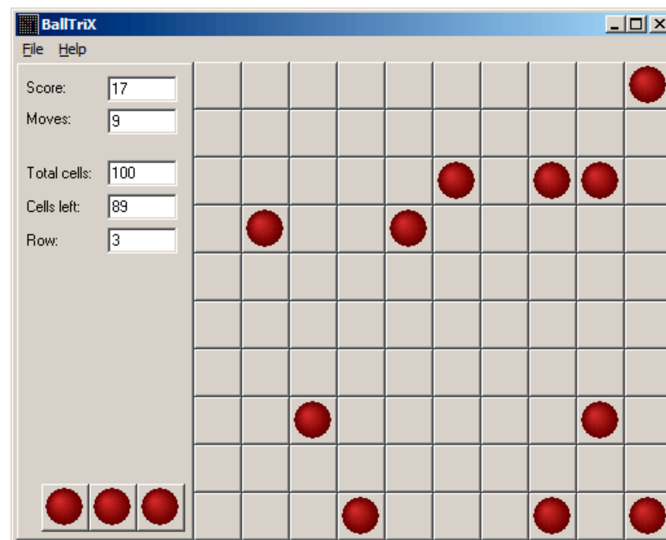


Fig. 8.6: La blague fonctionne

Hé oui, ça fonctionne[5].

Mais pourquoi est-ce que les arguments de la fonction `random()` sont des variables globales? C'est seulement parce qu'il est possible de changer la taille du plateau dans les préférences du jeu, donc ces valeurs ne sont pas codées en dur. Le valeurs 10 et 5 sont celles par défaut.

8.4 Démineur (Windows XP)

Pour ceux qui ne sont pas très bons avec le jeu démineur, nous pouvons essayer de révéler les mines cachées dans le débogueur.

Comme on le sait, le démineur place des mines aléatoirement, donc il doit y avoir une sorte de générateur de nombre aléatoire ou un appel à la fonction C standard `rand()`.

Ce qui est vraiment cool en rétro-ingénierant des produits Microsoft c'est qu'il y a les fichiers PDB avec les symboles (nom de fonctions, etc.) Lorsque nous chargeons `winmine.exe` dans IDA, il télécharge le fichier PDB exact pour cet exécutable et affiche tous les noms.

Donc le voici, le seul appel à `rand()` est cette fonction:

```
.text:01003940 ; __stdcall Rnd(x)
.text:01003940 _Rnd@4          proc near               ; CODE XREF:
    StartGame()+53
```

[5] J'ai fait une fois cette blague à des collègues dans l'espoir qu'ils arrêtent de jouer. Mais ça n'a pas fonctionné.

```
.text:01003940                                          ; StartGame()+61
.text:01003940
.text:01003940 arg_0           = dword ptr  4
.text:01003940
.text:01003940                 call    ds:__imp__rand
.text:01003946                 cdq
.text:01003947                 idiv    [esp+arg_0]
.text:0100394B                 mov     eax, edx
.text:0100394D                 retn    4
.text:0100394D _Rnd@4          endp
```

IDA l'a appelé ainsi, et c'est le nom que lui ont donné les développeurs du démineur.

La fonction est très simple:

```
int Rnd(int limit)
{
    return rand() % limit;
};
```

(Il n'y a pas de nom «limit» dans le fichier PDB ; nous avons nommé manuellement les arguments comme ceci.)

Donc elle renvoie une valeur aléatoire entre 0 et la limite spécifiée.

`Rnd()` est appelée depuis un seul endroit, la fonction appelée `StartGame()`, et il semble bien que ce soit exactement le code qui place les mines:

```
.text:010036C7                 push    _xBoxMac
.text:010036CD                 call    _Rnd@4          ; Rnd(x)
.text:010036D2                 push    _yBoxMac
.text:010036D8                 mov     esi, eax
.text:010036DA                 inc     esi
.text:010036DB                 call    _Rnd@4          ; Rnd(x)
.text:010036E0                 inc     eax
.text:010036E1                 mov     ecx, eax
.text:010036E3                 shl     ecx, 5          ; ECX=ECX*32
.text:010036E6                 test    _rgBlk[ecx+esi], 80h
.text:010036EE                 jnz     short loc_10036C7
.text:010036F0                 shl     eax, 5          ; EAX=EAX*32
.text:010036F3                 lea     eax, _rgBlk[eax+esi]
.text:010036FA                 or      byte ptr [eax], 80h
.text:010036FD                 dec     _cBombStart
.text:01003703                 jnz     short loc_10036C7
```

Le démineur vous permet de définir la taille du plateau, donc les dimensions X (xBoxMac) et Y (yBoxMac) du plateau sont des variables globales. Elles sont passées à `Rnd()` et des coordonnées aléatoires sont générées. Une mine est placée par l'instruction `OR` en `0x010036FA`. Et si une mine y a déjà été placée avant (il est possible que la fonction `Rnd()` génère une paire de coordonnées qui a déjà été générée), alors les instructions `TEST` et `JNZ` en `0x010036E6` bouclent sur la routine de génération.

`cBombStart` est la variable globale contenant le nombre total de mines. Donc ceci est une boucle.

La largeur du tableau est 32 (nous pouvons conclure ceci en regardant l'instruction SHL, qui multiplie l'une des coordonnées par 32).

La taille du tableau global `rgBlk` peut facilement être déduite par la différence entre le label `rgBlk` dans le segment de données et le label suivant. Il s'agit de 0x360 (864) :

```
.data:01005340 _rgBlk          db 360h dup(?)          ; DATA XREF:
    MainWndProc(x,x,x,x)+574
.data:01005340                                         ; DisplayBlk(x,x)+23
.data:010056A0 _Preferences    dd ?                    ; DATA XREF:
    FixMenus()+2
...
```

$864/32 = 27$.

Donc, la taille du tableau est-elle $27*32$? C'est proche de ce que nous savons: lorsque nous essayons de définir la taille du plateau à $100*100$ dans les préférences du démineur, il corrige à une taille de plateau de $24*30$. Donc ceci est la taille maximale du plateau. Et le tableau a une taille fixe, pour toutes les tailles de plateau.

REgardons tout ceci dans OllyDbg. Nous allons lancer le démineur, lui attacher OllyDbg et nous allons pouvoir voir le contenu de la mémoire à l'adresse du tableau `rgBlk` (`0x01005340`)[6]. Donc nous avons ceci à l'emplacement mémoire du tableau:

```
Address   Hex dump
01005340  10 10 10 10|10 10 10 10|10 10 10 0F|0F 0F 0F 0F|
01005350  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005360  10 0F 0F 0F|0F 0F 0F 0F|0F 0F 10 0F|0F 0F 0F 0F|
01005370  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005380  10 0F 0F 0F|0F 0F 0F 0F|0F 0F 10 0F|0F 0F 0F 0F|
01005390  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010053A0  10 0F 0F 0F|0F 0F 0F 0F|8F 0F 10 0F|0F 0F 0F 0F|
010053B0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010053C0  10 0F 0F 0F|0F 0F 0F 0F|0F 0F 10 0F|0F 0F 0F 0F|
010053D0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010053E0  10 0F 0F 0F|0F 0F 0F 0F|0F 0F 10 0F|0F 0F 0F 0F|
010053F0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005400  10 0F 0F 8F|0F 0F 8F 0F|0F 0F 10 0F|0F 0F 0F 0F|
01005410  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005420  10 8F 0F 0F|8F 0F 0F 0F|0F 0F 10 0F|0F 0F 0F 0F|
01005430  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005440  10 8F 0F 0F|0F 0F 8F 0F|0F 8F 10 0F|0F 0F 0F 0F|
01005450  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005460  10 0F 0F 0F|0F 8F 0F 0F|0F 8F 10 0F|0F 0F 0F 0F|
01005470  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005480  10 10 10 10|10 10 10 10|10 10 10 0F|0F 0F 0F 0F|
01005490  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010054A0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010054B0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010054C0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
```

[6]Toutes les adresses ici sont pour le démineur de Windows XP SP3 English. Elles peuvent être différentes pour d'autres services packs.

OllyDbg, comme tout autre éditeur hexadécimal, affiche 16 octets par ligne. Donc chaque ligne de tableau de 32-octet occupe exactement 2 lignes ici.

Ceci est le niveau débutant (plateau de 9*9).

Il y a une sorte de structure carré que l'on voit ici (octets 0x10).

Nous cliquons «Run» dans OllyDbg pour débloquer le processus du démineur, puis nous cliquons au hasard dans la fenêtre du démineur et nous tombons sur une mine, mais maintenant, toutes les mines sont visibles:

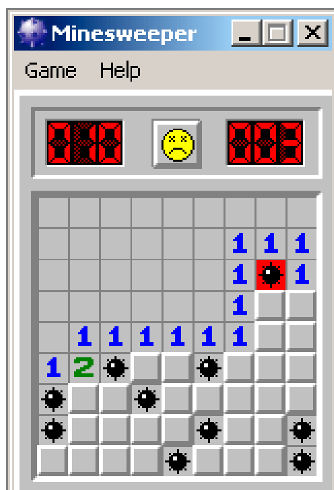


Fig. 8.7: Mines

En comparant les emplacements des mines et le dump, nous pouvons en conclure que 0x10 correspond au bord, 0x0F—bloc vide, 0x8F—mine. Peut-être que 0x10 est simplement une *valeur sentinelle*.

Maintenant nous allons ajouter des commentaires et aussi mettre tous les octets à 0x8F entre parenthèses droites:

```
border:
01005340  10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005350  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #1:
01005360  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005370  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #2:
01005380  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005390  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #3:
010053A0  10 0F 0F 0F 0F 0F 0F 0F[8F]0F 10 0F 0F 0F 0F 0F
010053B0  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #4:
010053C0  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
010053D0  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #5:
```

```
010053E0  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
010053F0  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #6:
01005400  10 0F 0F[8F]0F 0F[8F]0F 0F 0F 10 0F 0F 0F 0F 0F
01005410  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #7:
01005420  10[8F]0F 0F[8F]0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005430  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #8:
01005440  10[8F]0F 0F 0F 0F[8F]0F 0F[8F]10 0F 0F 0F 0F 0F
01005450  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #9:
01005460  10 0F 0F 0F 0F[8F]0F 0F 0F[8F]10 0F 0F 0F 0F 0F
01005470  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
border:
01005480  10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005490  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
```

Maintenant nous allons supprimer tous les *octet de bord* (0x10) et ce qu'il y a après:

```
 0F 0F 0F 0F 0F 0F 0F 0F 0F
 0F 0F 0F 0F 0F 0F 0F 0F 0F
 0F 0F 0F 0F 0F 0F 0F[8F]0F
 0F 0F 0F 0F 0F 0F 0F 0F 0F
 0F 0F 0F 0F 0F 0F 0F 0F 0F
 0F 0F[8F]0F 0F[8F]0F 0F 0F
[8F]0F 0F[8F]0F 0F 0F 0F 0F
[8F]0F 0F 0F 0F[8F]0F 0F[8F]
 0F 0F 0F 0F[8F]0F 0F 0F[8F]
```

Oui, ce sont des mines, maintenant ça peut être vu clairement et comparé avec la copie d'écran.

Ce qui est intéressant, c'est que nous pouvons modifier le tableau directement dans OllyDbg. Nous pouvons supprimer toutes les mines en changeant les octets à 0x8F par 0x0F, et voici ce que nous obtenons dans le démineur:

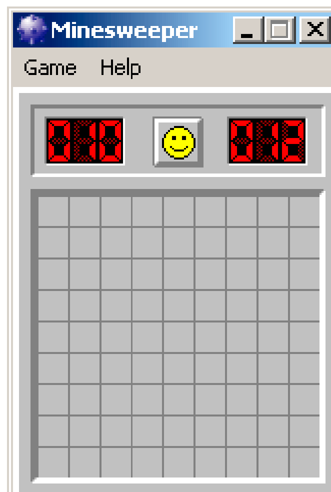


Fig. 8.8: Toutes les mines sont supprimées depuis le débogueur

Nous pouvons aussi toutes les déplacer à la première ligne:

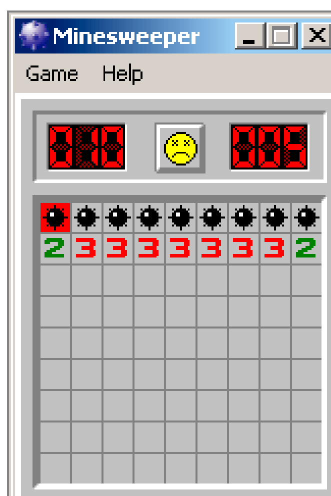


Fig. 8.9: Mines mises dans le débogueur

Bon, le débogueur n'est pas très pratique pour espionner (ce qui est notre but), donc nous allons écrire un petit utilitaire pour afficher le contenu du plateau:

```
// Windows XP MineSweeper cheater
// written by dennis(a)yurichev.com for http://beginners.re/ book
#include <windows.h>
```

```
#include <assert.h>
#include <stdio.h>

int main (int argc, char * argv[])
{
        int i, j;
        HANDLE h;
        DWORD PID, address, rd;
        BYTE board[27][32];

        if (argc!=3)
        {
                printf ("Usage: %s <PID> <address>\n", argv[0]);
                return 0;
        };

        assert (argv[1]!=NULL);
        assert (argv[2]!=NULL);

        assert (sscanf (argv[1], "%d", &PID)==1);
        assert (sscanf (argv[2], "%x", &address)==1);

        h=OpenProcess (PROCESS_VM_OPERATION | PROCESS_VM_READ | ↙
    ↘ PROCESS_VM_WRITE, FALSE, PID);

        if (h==NULL)
        {
                DWORD e=GetLastError();
                printf ("OpenProcess error: %08X\n", e);
                return 0;
        };

        if (ReadProcessMemory (h, (LPVOID)address, board, sizeof(board), &↙
    ↘ rd)!=TRUE)
        {
                printf ("ReadProcessMemory() failed\n");
                return 0;
        };

        for (i=1; i<26; i++)
        {
                if (board[i][0]==0x10 && board[i][1]==0x10)
                        break; // end of board
                for (j=1; j<31; j++)
                {
                        if (board[i][j]==0x10)
                                break; // board border
                        if (board[i][j]==0x8F)
                                printf ("*");
                        else
                                printf (" ");

                };
```

```
                printf ("\n");
        };

        CloseHandle (h);
};
```

Simplement donner le PID[7] [8] et l'adresse du tableau (0x01005340 pour Windows XP SP3 English) et il l'affichera [9].

Il s'attache à un processus win32 par le PID et lit la mémoire du processus à l'adresse.

8.4.1 Trouver la grille automatiquement

C'est pénible de mettre l'adresse à chaque fois que nous lançons notre utilitaire. Aussi, différentes versions du démineur peuvent avoir le tableau à des adresses différentes. Sachant qu'il a toujours un bord (octets 0x10), nous pouvons le trouver facilement en mémoire:

```
        // find frame to determine the address
        process_mem=(BYTE*)malloc(process_mem_size);
        assert (process_mem!=NULL);

        if (ReadProcessMemory (h, (LPVOID)start_addr, process_mem, ↙
    ↘ process_mem_size, &rd)!=TRUE)
        {
                printf ("ReadProcessMemory() failed\n");
                return 0;
        };

        // for 9*9 grid.
        // FIXME: slow!
        for (i=0; i<process_mem_size; i++)
        {
                if (memcmp(process_mem+i, "\x10\x10\x10\x10\x10\x10\x10\x10↙
    ↘ \x10\x10\x10\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\↙
    ↘ x0F\x0F\x0F\x0F\x0F\x0F\x0F\x10", 32)==0)
                {
                        // found
                        address=start_addr+i;
                        break;
                };
        };
        if (address==0)
        {
                printf ("Can't determine address of frame (and grid)\n");
                return 0;
        }
        else
        {
```

[7]ID d'un processus
[8]Le PID peut être vu dans le Task Manager (l'activer avec «View → Select Columns»)
[9]L'exécutable compilé est ici: beginners.re

```
                printf ("Found frame and grid at 0x%x\n", address);
        };
```

Code source complet: https://beginners.re/paywall/RE4B-source/current-tree//examples/minesweeper/minesweeper_cheater2.c.

8.4.2 Exercices

- Pourquoi est-ce que les *octets de bord* (ou *valeurs sentinelles*) (0x10) existent dans le tableau?

  À quoi servent-elles si elles ne sont pas visibles dans l'interface du démineur? Comment est-ce qu'il pourrait fonctionner sans elles?
- Comme on s'en doute, il y a plus de valeurs possible (pour les blocs ouverts, ceux flagués par l'utilisateur, etc.). Essayez de trouver la signification de chacune d'elles.
- Modifiez mon utilitaire afin qu'il puisse supprimer toutes les mines ou qu'il les place suivant un schéma fixé de votre choix dans le démineur.

8.5 Hacker l'horloge de Windows

Parfois je fais des poissons d'avril à mes collègues.

Cherchons si nous pourrions faire quelque chose avec l'horloge de Windows? Pouvons-nous la forcer à tourner à l'envers?

Tout d'abord, lorsque l'on clique sur date/time dans la barre d'état,
le module *C:\WINDOWS\SYSTEM32\TIMEDATE.CPL* est exécuté, qui est un fichier exécutable PE habituel.

Voyons d'abord comment il affiche les aiguilles. Lorsque j'ouvre le fichier (de Windows 7) dans Resource Hacker, il y a le fond de l'horloge, mais sans aiguille:

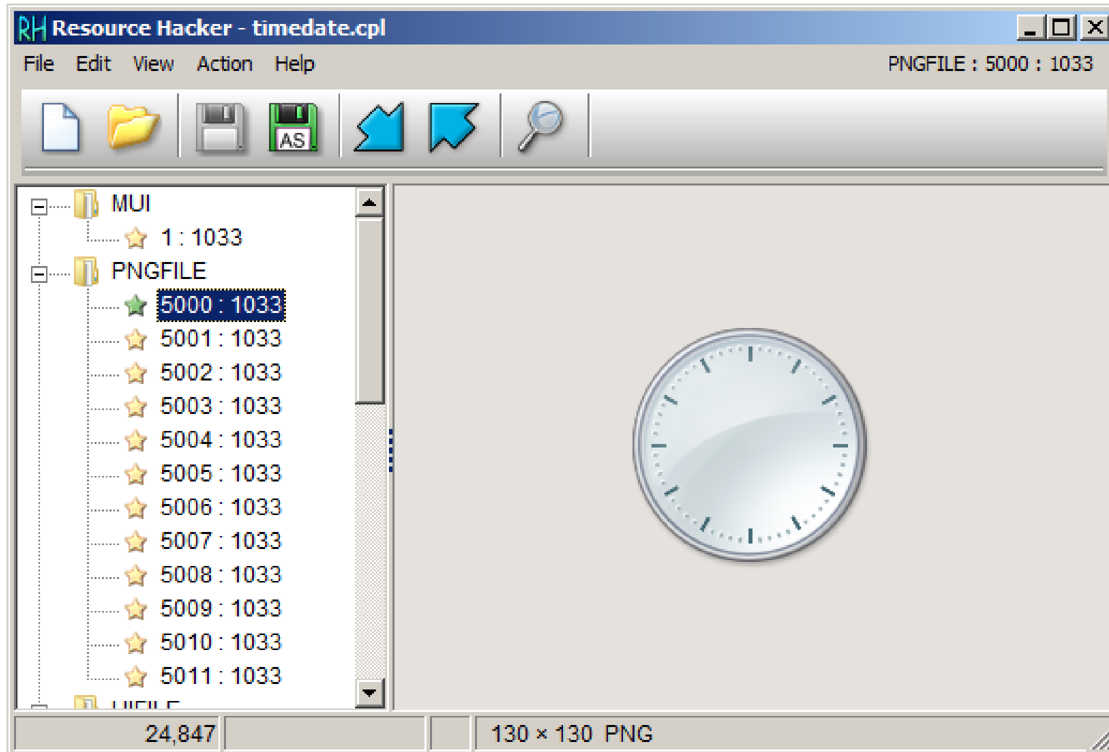


Fig. 8.10: Resource Hacker

Ok, que savons-nous? Comment afficher une aiguille? Elles commencent au milieu du cercle, s'arrêtent sur son bord. De ce fait, nous devons calculer les coordonnées d'un point sur le bord d'un cercle. Des mathématiques scolaires, nous pouvons nous rappeler que nous devons utiliser les fonctions sinus/cosinus pour dessiner un cercle, ou au moins la racine carré. Il n'y a pas de telles choses dans *TIMEDATE.CPL*, au moins à première vue. Mais grâce au fichier PDB de débogage de Microsoft, je peux trouver une fonction appelée *CAnalogClock::DrawHand()*, qui appelle *Gdiplus::Graphics::DrawLine()* au moins deux fois.

Voici le code:

```
.text:6EB9DBC7 ; private: enum Gdiplus::Status __thiscall
    CAnalogClock::_DrawHand(class Gdiplus::Graphics *, int, struct ClockHand
    const &, class Gdiplus::Pen *)
.text:6EB9DBC7 ?_DrawHand@CAnalogClock@@AAE?
    AW4Status@Gdiplus@@PAVGraphics@3@HABUClockHand@@PAVPen@3@@Z proc near
.text:6EB9DBC7       ; CODE XREF: CAnalogClock::_ClockPaint(HDC__ *)+163
.text:6EB9DBC7       ; CAnalogClock::_ClockPaint(HDC__ *)+18B
.text:6EB9DBC7
.text:6EB9DBC7 var_10          = dword ptr -10h
.text:6EB9DBC7 var_C           = dword ptr -0Ch
.text:6EB9DBC7 var_8           = dword ptr -8
.text:6EB9DBC7 var_4           = dword ptr -4
```

```
.text:6EB9DBC7 arg_0           = dword ptr  8
.text:6EB9DBC7 arg_4           = dword ptr  0Ch
.text:6EB9DBC7 arg_8           = dword ptr  10h
.text:6EB9DBC7 arg_C           = dword ptr  14h
.text:6EB9DBC7
.text:6EB9DBC7                 mov     edi, edi
.text:6EB9DBC9                 push    ebp
.text:6EB9DBCA                 mov     ebp, esp
.text:6EB9DBCC                 sub     esp, 10h
.text:6EB9DBCF                 mov     eax, [ebp+arg_4]
.text:6EB9DBD2                 push    ebx
.text:6EB9DBD3                 push    esi
.text:6EB9DBD4                 push    edi
.text:6EB9DBD5                 cdq
.text:6EB9DBD6                 push    3Ch
.text:6EB9DBD8                 mov     esi, ecx
.text:6EB9DBDA                 pop     ecx
.text:6EB9DBDB                 idiv    ecx
.text:6EB9DBDD                 push    2
.text:6EB9DBDF                 lea     ebx, table[edx*8]
.text:6EB9DBE6                 lea     eax, [edx+1Eh]
.text:6EB9DBE9                 cdq
.text:6EB9DBEA                 idiv    ecx
.text:6EB9DBEC                 mov     ecx, [ebp+arg_0]
.text:6EB9DBEF                 mov     [ebp+var_4], ebx
.text:6EB9DBF2                 lea     eax, table[edx*8]
.text:6EB9DBF9                 mov     [ebp+arg_4], eax
.text:6EB9DBFC                 call    ?↙
    ↘ SetInterpolationMode@Graphics@Gdiplus@@QAE?↙
    ↘ AW4Status@2@W4InterpolationMode@2@@Z ;
    Gdiplus::Graphics::SetInterpolationMode(Gdiplus::InterpolationMode)
.text:6EB9DC01                 mov     eax, [esi+70h]
.text:6EB9DC04                 mov     edi, [ebp+arg_8]
.text:6EB9DC07                 mov     [ebp+var_10], eax
.text:6EB9DC0A                 mov     eax, [esi+74h]
.text:6EB9DC0D                 mov     [ebp+var_C], eax
.text:6EB9DC10                 mov     eax, [edi]
.text:6EB9DC12                 sub     eax, [edi+8]
.text:6EB9DC15                 push    8000            ; nDenominator
.text:6EB9DC1A                 push    eax             ; nNumerator
.text:6EB9DC1B                 push    dword ptr [ebx+4] ; nNumber
.text:6EB9DC1E                 mov     ebx, ds:__imp__MulDiv@12 ;
    MulDiv(x,x,x)
.text:6EB9DC24                 call    ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC26                 add     eax, [esi+74h]
.text:6EB9DC29                 push    8000            ; nDenominator
.text:6EB9DC2E                 mov     [ebp+arg_8], eax
.text:6EB9DC31                 mov     eax, [edi]
.text:6EB9DC33                 sub     eax, [edi+8]
.text:6EB9DC36                 push    eax             ; nNumerator
.text:6EB9DC37                 mov     eax, [ebp+var_4]
.text:6EB9DC3A                 push    dword ptr [eax] ; nNumber
.text:6EB9DC3C                 call    ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC3E                 add     eax, [esi+70h]
```

```
.text:6EB9DC41                 mov     ecx, [ebp+arg_0]
.text:6EB9DC44                 mov     [ebp+var_8], eax
.text:6EB9DC47                 mov     eax, [ebp+arg_8]
.text:6EB9DC4A                 mov     [ebp+var_4], eax
.text:6EB9DC4D                 lea     eax, [ebp+var_8]
.text:6EB9DC50                 push    eax
.text:6EB9DC51                 lea     eax, [ebp+var_10]
.text:6EB9DC54                 push    eax
.text:6EB9DC55                 push    [ebp+arg_C]
.text:6EB9DC58                 call    ?DrawLine@Graphics@Gdiplus@@QAE?↙
    ↘ AW4Status@2@PBVPen@2@ABVPoint@2@1@Z ;
    Gdiplus::Graphics::DrawLine(Gdiplus::Pen const *,Gdiplus::Point const
    &,Gdiplus::Point const &)
.text:6EB9DC5D                 mov     ecx, [edi+8]
.text:6EB9DC60                 test    ecx, ecx
.text:6EB9DC62                 jbe     short loc_6EB9DCAA
.text:6EB9DC64                 test    eax, eax
.text:6EB9DC66                 jnz     short loc_6EB9DCAA
.text:6EB9DC68                 mov     eax, [ebp+arg_4]
.text:6EB9DC6B                 push    8000            ; nDenominator
.text:6EB9DC70                 push    ecx             ; nNumerator
.text:6EB9DC71                 push    dword ptr [eax+4] ; nNumber
.text:6EB9DC74                 call    ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC76                 add     eax, [esi+74h]
.text:6EB9DC79                 push    8000            ; nDenominator
.text:6EB9DC7E                 push    dword ptr [edi+8] ; nNumerator
.text:6EB9DC81                 mov     [ebp+arg_8], eax
.text:6EB9DC84                 mov     eax, [ebp+arg_4]
.text:6EB9DC87                 push    dword ptr [eax] ; nNumber
.text:6EB9DC89                 call    ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC8B                 add     eax, [esi+70h]
.text:6EB9DC8E                 mov     ecx, [ebp+arg_0]
.text:6EB9DC91                 mov     [ebp+var_8], eax
.text:6EB9DC94                 mov     eax, [ebp+arg_8]
.text:6EB9DC97                 mov     [ebp+var_4], eax
.text:6EB9DC9A                 lea     eax, [ebp+var_8]
.text:6EB9DC9D                 push    eax
.text:6EB9DC9E                 lea     eax, [ebp+var_10]
.text:6EB9DCA1                 push    eax
.text:6EB9DCA2                 push    [ebp+arg_C]
.text:6EB9DCA5                 call    ?DrawLine@Graphics@Gdiplus@@QAE?↙
    ↘ AW4Status@2@PBVPen@2@ABVPoint@2@1@Z ;
    Gdiplus::Graphics::DrawLine(Gdiplus::Pen const *,Gdiplus::Point const
    &,Gdiplus::Point const &)
.text:6EB9DCAA
.text:6EB9DCAA loc_6EB9DCAA:   ; CODE XREF:
    CAnalogClock::_DrawHand(Gdiplus::Graphics *,int,ClockHand const
    &,Gdiplus::Pen *)+9B
.text:6EB9DCAA                 ; CAnalogClock::_DrawHand(Gdiplus::Graphics
    *,int,ClockHand const &,Gdiplus::Pen *)+9F
.text:6EB9DCAA                 pop     edi
.text:6EB9DCAB                 pop     esi
.text:6EB9DCAC                 pop     ebx
.text:6EB9DCAD                 leave
.text:6EB9DCAE                 retn    10h
```

```
.text:6EB9DCAE ?_DrawHand@CAnalogClock@@AAE?↙
    ↘ AW4Status@Gdiplus@@PAVGraphics@3@HABUClockHand@@PAVPen@3@@Z endp
.text:6EB9DCAE
```

Nous voyons que les arguments de *DrawLine()* dépendent du résultat de la fonction *MulDiv()* et d'une table *table[]* (le nom est mien), qui a des éléments de 8-octets (regardez le second opérande de LEA).

Qu'y a-t-il dans table[]?

```
.text:6EB87890 ; int table[]
.text:6EB87890 table           dd 0
.text:6EB87894                 dd 0FFFFE0C1h
.text:6EB87898                 dd 344h
.text:6EB8789C                 dd 0FFFFE0ECh
.text:6EB878A0                 dd 67Fh
.text:6EB878A4                 dd 0FFFFE16Fh
.text:6EB878A8                 dd 9A8h
.text:6EB878AC                 dd 0FFFFE248h
.text:6EB878B0                 dd 0CB5h
.text:6EB878B4                 dd 0FFFFE374h
.text:6EB878B8                 dd 0F9Fh
.text:6EB878BC                 dd 0FFFFE4F0h
.text:6EB878C0                 dd 125Eh
.text:6EB878C4                 dd 0FFFFE6B8h
.text:6EB878C8                 dd 14E9h

...
```

Elle n'est référencée que depuis la fonction *DrawHand()*. Elle a 120 mots de 32-bit ou 60 paires 32-bit... attendez, 60? Regardons ces valeurs de plus près. Tout d'abord, je vais remplacer 6 paires ou 12 mots de 32-bit par des zéros, et je vais mettre le fichier *TIMEDATE.CPL* modifié dans *C:\WINDOWS\SYSTEM32*. (Vous pourriez devoir changer le propriétaire du fichier *TIMEDATE.CPL* pour votre compte utilisateur primaire (au lieu de *TrustedInstaller*), et donc, démarrer en mode sans échec avec la ligne de commande afin de pouvoir copier le fichier, qui est en général bloqué.)

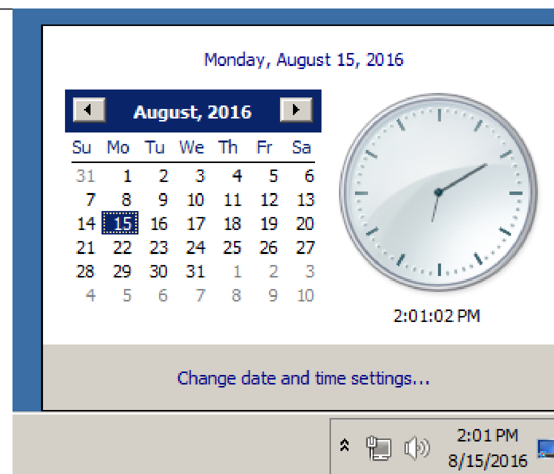


Fig. 8.11: Tentative d'exécution

Maintenant lorsqu'une aiguilles est située dans 0..5 secondes/minutes, elle est invisible! Toutefois, la partie opposée (plus courte) de la seconde aiguille est visible et bouge. Lorsqu'une aiguille est en dehors de cette partie, elle est visible comme d'habitude.

Regardons d' encore plus près la table dans Mathematica. J'ai copié/collé la table de *TIMEDATE.CPL* dans un fichier *tbl* (480 octets). Nous tenons pour acquis le fait que ce sont des valeurs signées, car la moitié des éléments sont inférieurs à zéro (0FFFFE0C1h, etc.). Si ces valeurs étaient non signées, elles seraient étrangement grandes.

```
In[]:= tbl = BinaryReadList["~/.../tbl", "Integer32"]

Out[]= {0, -7999, 836, -7956, 1663, -7825, 2472, -7608, 3253, -7308, 3999, ↙
    ↳ \
-6928, 4702, -6472, 5353, -5945, 5945, -5353, 6472, -4702, 6928, \
-4000, 7308, -3253, 7608, -2472, 7825, -1663, 7956, -836, 8000, 0, \
7956, 836, 7825, 1663, 7608, 2472, 7308, 3253, 6928, 4000, 6472, \
4702, 5945, 5353, 5353, 5945, 4702, 6472, 3999, 6928, 3253, 7308, \
2472, 7608, 1663, 7825, 836, 7956, 0, 7999, -836, 7956, -1663, 7825, \
-2472, 7608, -3253, 7308, -4000, 6928, -4702, 6472, -5353, 5945, \
-5945, 5353, -6472, 4702, -6928, 3999, -7308, 3253, -7608, 2472, \
-7825, 1663, -7956, 836, -7999, 0, -7956, -836, -7825, -1663, -7608, \
-2472, -7308, -3253, -6928, -4000, -6472, -4702, -5945, -5353, -5353, \
-5945, -4702, -6472, -3999, -6928, -3253, -7308, -2472, -7608, -1663, \
-7825, -836, -7956}

In[]:= Length[tbl]
Out[]= 120
```

Traitons deux valeurs consécutives comme une paire:

```
In[]:= pairs = Partition[tbl, 2]
```

```
Out[]= {{0, -7999}, {836, -7956}, {1663, -7825}, {2472, -7608}, \
{3253, -7308}, {3999, -6928}, {4702, -6472}, {5353, -5945}, {5945, \
-5353}, {6472, -4702}, {6928, -4000}, {7308, -3253}, {7608, -2472}, \
{7825, -1663}, {7956, -836}, {8000, 0}, {7956, 836}, {7825,
1663}, {7608, 2472}, {7308, 3253}, {6928, 4000}, {6472,
4702}, {5945, 5353}, {5353, 5945}, {4702, 6472}, {3999,
6928}, {3253, 7308}, {2472, 7608}, {1663, 7825}, {836, 7956}, {0,
7999}, {-836, 7956}, {-1663, 7825}, {-2472, 7608}, {-3253,
7308}, {-4000, 6928}, {-4702, 6472}, {-5353, 5945}, {-5945,
5353}, {-6472, 4702}, {-6928, 3999}, {-7308, 3253}, {-7608,
2472}, {-7825, 1663}, {-7956, 836}, {-7999,
0}, {-7956, -836}, {-7825, -1663}, {-7608, -2472}, {-7308, -3253}, \
{-6928, -4000}, {-6472, -4702}, {-5945, -5353}, {-5353, -5945}, \
{-4702, -6472}, {-3999, -6928}, {-3253, -7308}, {-2472, -7608}, \
{-1663, -7825}, {-836, -7956}}

In[]:= Length[pairs]
Out[]= 60
```

Essayons de traiter chaque paire comme des coordonnées X/Y et dessinons les 60 paires, et aussi les 15 premières paires:

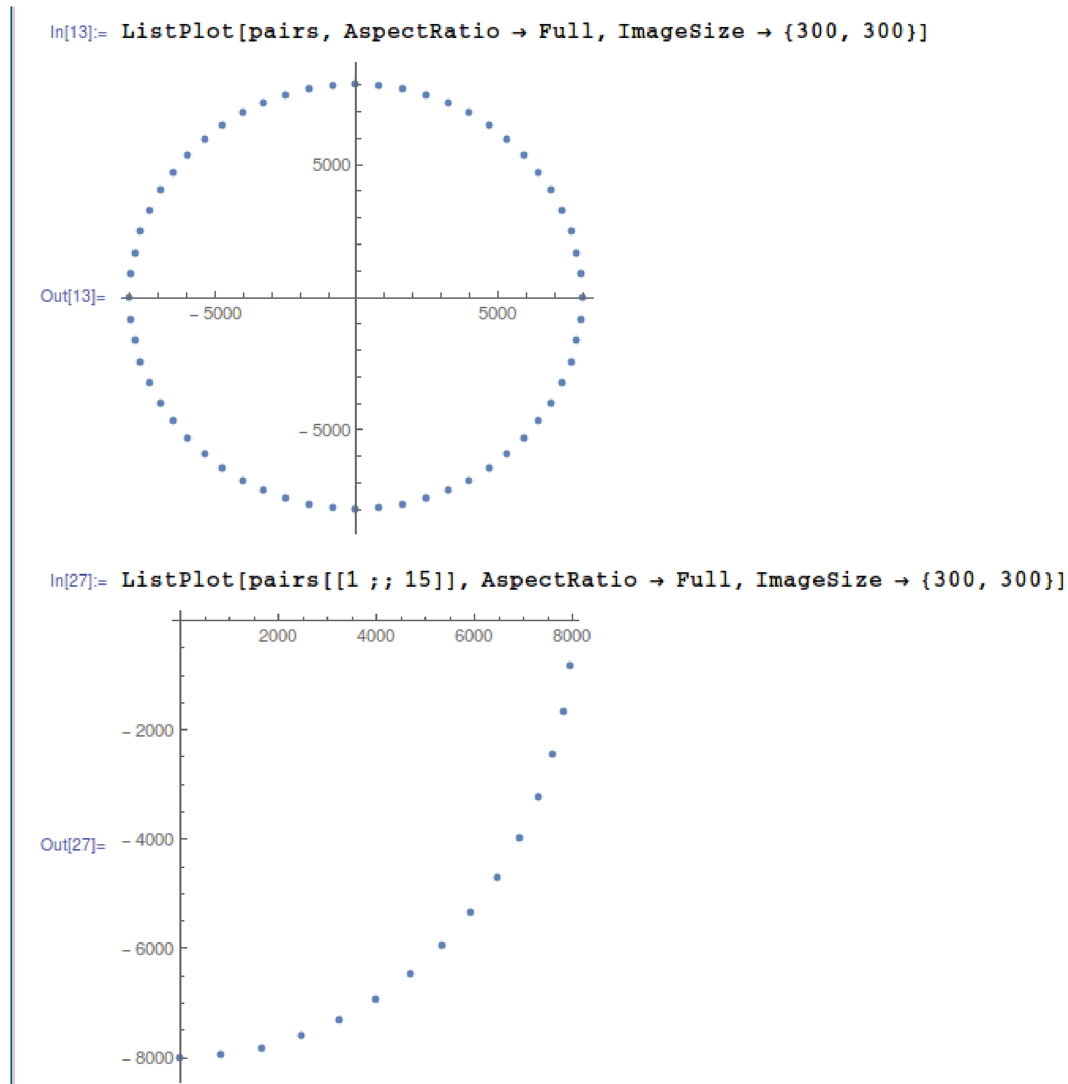


Fig. 8.12: Mathematica

Ça donne quelque chose! Chaque paire est juste une coordonnée. Les 15 premières paires sont les coordonnées pour $\frac{1}{4}$ de cercle.

Peut-être que les développeurs de Microsoft ont pré-calculé toutes les coordonnées et les ont mises dans une table. myindexMemoization Ceci est une pratique très répandue, quoique désuète – l'accès à une table précalculée est plus rapide que d'appeler les fonctions sinus/cosinus relativement lente[10]. Les opérations sinus/cosinus ne sont plus aussi couteuses...

Maintenant, je comprends pourquoi lorsque j'ai effacé les 6 premières paires, les

[10]Aujourd'hui ceci est appelé la *memoïsation*

aiguilles étaient invisibles dans cette zone: en fait, les aiguilles étaient dessinées, elles avaient juste une longueur de zéro, car elles commençaient et finissaient en (0,0).

La blague

Sachant tout cela, comment serait-il possible de forcer les aiguilles à tourner à l'envers? En fait, ceci est simple, nous devons seulement tourner la table, afin que chaque aiguille, au lieu d'être dessinée à l'index 0, le soit à l'index 59.

J'ai créé le modificateur il y a longtemps, au tout début des années 2000, pour Windows 2000. Difficile à croire, il fonctionne toujours pour Windows 7, peut-être que la table n'a pas changé depuis lors!

Code source du modificateur: https://beginners.re/paywall/RE4B-source/current-tree//examples/timedate/time_pt.c.

Maintenant, je peux voir les aiguilles tourner à l'envers:

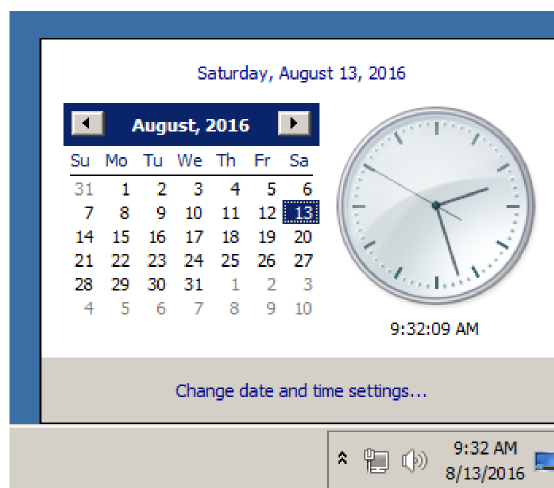


Fig. 8.13: Maintenant ça fonctionne

Bon, il n'y a pas d'animation dans ce livre, mais si vous y regardez de plus près, vous pouvez voir que les aiguilles affichent en fait l'heure correcte, mais que la surface entière de l'horloge est tournée verticalement, comme si nous la voyons depuis l'intérieur de l'horloge.

Code source divulgué de Windows 2000

Donc, j'ai écrit le modificateur et ensuite le code source de Windows 2000 a fuité (je ne peux toutefois pas vous obligez à me croire). Jettons un coup d'œil au code source de cette fonction et à la table.
Le fichier est *win2k/private/shell/cpls/utc/clock.c* :

```
//
```

```
// Array containing the sine and cosine values for hand positions.
//
POINT rCircleTable[] =
{
    { 0,     -7999},
    { 836,   -7956},
    { 1663,  -7825},
    { 2472,  -7608},
    { 3253,  -7308},
...
    { -4702, -6472},
    { -3999, -6928},
    { -3253, -7308},
    { -2472, -7608},
    { -1663, -7825},
    { -836 , -7956},
};

////////////////////////////////////////////////////////////////////////////
//
// DrawHand
//
// Draws the hands of the clock.
//
////////////////////////////////////////////////////////////////////////////

void DrawHand(
    HDC hDC,
    int pos,
    HPEN hPen,
    int scale,
    int patMode,
    PCLOCKSTR np)
{
    LPPOINT lppt;
    int radius;

    MoveTo(hDC, np->clockCenter.x, np->clockCenter.y);
    radius = MulDiv(np->clockRadius, scale, 100);
    lppt = rCircleTable + pos;
    SetROP2(hDC, patMode);
    SelectObject(hDC, hPen);

    LineTo( hDC,
            np->clockCenter.x + MulDiv(lppt->x, radius, 8000),
            np->clockCenter.y + MulDiv(lppt->y, radius, 8000) );
}
```

Maintenant, c'est clair: les coordonnées sont pré-calculées comme si la surface de l'horloge avait une hauteur et une largeur de $2 \cdot 8000$, et ensuite elles sont adaptées au rayon actuel de l'horloge en utilisant la fonction *MulDiv()*.

La structure POINT[11] est une structure de deux valeurs 32-bit, la première est *x*, la seconde *y*.

8.6 Solitaire (Windows 7) : blagues

8.6.1 51 cartes

Ceci est une blague que je fis une fois à mes collègues qui jouaient trop au jeu Solitaire. Je me demandais s'il était possible de supprimer quelques cartes, ou même en ajouter (dupliquer).

J'ai ouvert Solitaire.exe dans le dés-assembleur IDA, qui a demandé à télé-charger le fichier PDB depuis les serveurs de Microsoft. Ceci est habituellement la règle pour de nombreux exécutables et DLLs Windows. Au moins, le PDB contient tous les noms de fonctions.

Ensuite j'ai essayé de trouver le nombre 52 dans toutes les fonctions (car ce jeu de carte utilise 52 cartes). Il s'est avéré que seulement 2 fonctions l'avait.

La première est:

```
.text:00000001000393B4 ; __int64 __fastcall SolitaireGame::OnMoveComplete(↙
    ↘ SolitaireGame *this)
.text:00000001000393B4 ?OnMoveComplete@SolitaireGame@@QEAAHXZ proc near

...
```

La seconde est la fonction avec un nom significatif (nom tiré du PDB par IDA) : `InitialDeal()` :

```
.text:00000001000365F8 ; void __fastcall SolitaireGame::InitialDeal(↙
    ↘ SolitaireGame *__hidden this)
.text:00000001000365F8 ?InitialDeal@SolitaireGame@@QEAAXXZ proc near
.text:00000001000365F8
.text:00000001000365F8 var_58          = byte ptr -58h
.text:00000001000365F8 var_48          = qword ptr -48h
.text:00000001000365F8 var_40          = dword ptr -40h
.text:00000001000365F8 var_3C          = dword ptr -3Ch
.text:00000001000365F8 var_38          = dword ptr -38h
.text:00000001000365F8 var_30          = qword ptr -30h
.text:00000001000365F8 var_28          = xmmword ptr -28h
.text:00000001000365F8 var_18          = byte ptr -18h
.text:00000001000365F8
.text:00000001000365F8 ; FUNCTION CHUNK AT .text:00000001000A55C2 SIZE ↙
    ↘ 00000018 BYTES
.text:00000001000365F8
.text:00000001000365F8 ; __unwind { // __CxxFrameHandler3
.text:00000001000365F8                 mov     rax, rsp
.text:00000001000365FB                 push    rdi
.text:00000001000365FC                 push    r12
.text:00000001000365FE                 push    r13
```

[11] https://msdn.microsoft.com/en-us/library/windows/desktop/dd162805(v=vs.85).aspx

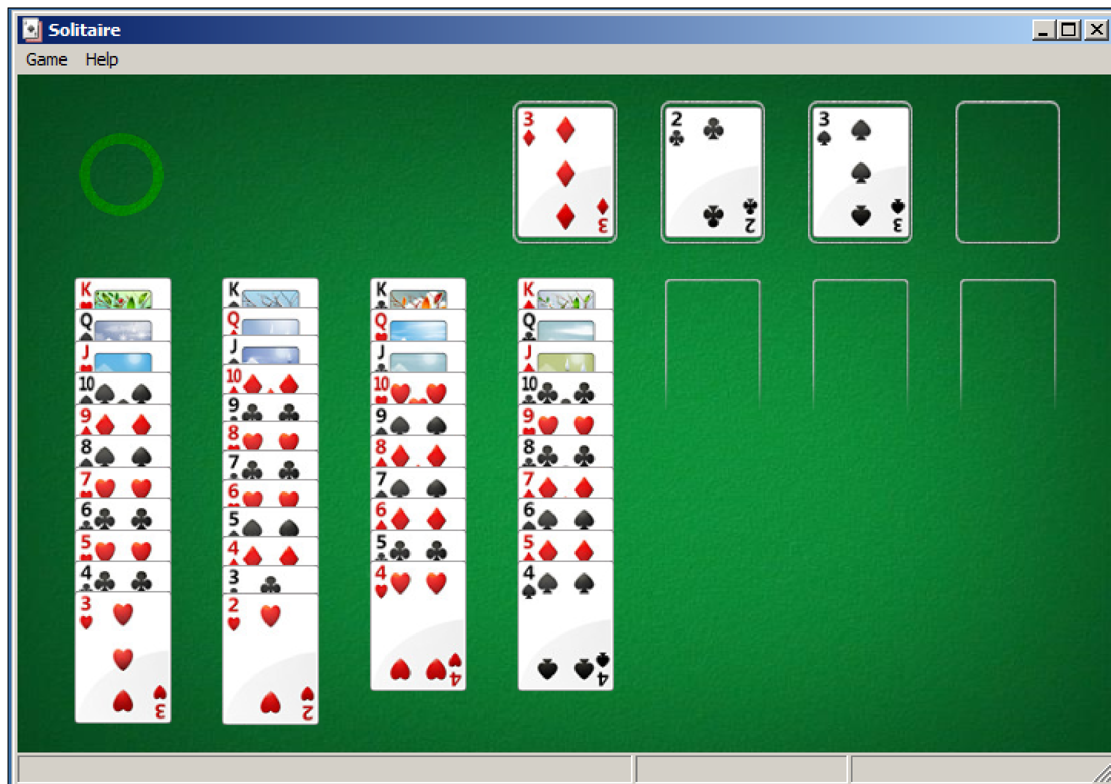
```
.text:0000000100036600                 sub     rsp, 60h
.text:0000000100036604                 mov     [rsp+78h+var_48], 0⤦
    ↳ FFFFFFFFFFFFFFFEh
.text:000000010003660D                 mov     [rax+8], rbx
.text:0000000100036611                 mov     [rax+10h], rbp
.text:0000000100036615                 mov     [rax+18h], rsi
.text:0000000100036619                 movaps  xmmword ptr [rax-28h], xmm6
.text:000000010003661D                 mov     rsi, rcx
.text:0000000100036620                 xor     edx, edx        ; struct ⤦
    ↳ Card *
.text:0000000100036622                 call    ?⤦
    ↳ SetSelectedCard@SolitaireGame@@QEAAXPEAVCard@@@Z ; SolitaireGame::⤦
    ↳ SetSelectedCard(Card *)
.text:0000000100036627                 and     qword ptr [rsi+0F0h], 0
.text:000000010003662F                 mov     rax, cs:?⤦
    ↳ g_pSolitaireGame@@3PEAVSolitaireGame@@EA ; SolitaireGame * ⤦
    ↳ g_pSolitaireGame
.text:0000000100036636                 mov     rdx, [rax+48h]
.text:000000010003663A                 cmp     byte ptr [rdx+51h], 0
.text:000000010003663E                 jz      short loc_10003664E
.text:0000000100036640                 xor     r8d, r8d        ; bool
.text:0000000100036643                 mov     dl, 1           ; int
.text:0000000100036645                 lea     ecx, [r8+3]     ; this
.text:0000000100036649                 call    ?⤦
    ↳ PlaySoundProto@GameAudio@@YA_NH_NPEAI@Z ; GameAudio::PlaySoundProto(⤦
    ↳ int,bool,uint *)
.text:000000010003664E
.text:000000010003664E loc_10003664E:                          ; CODE XREF:⤦
    ↳  SolitaireGame::InitialDeal(void)+46
.text:000000010003664E                 mov     rbx, [rsi+88h]
.text:0000000100036655                 mov     r8d, 4
.text:000000010003665B                 lea     rdx, aCardstackCreat ; "⤦
    ↳ CardStack::CreateDeck()::uiNumSuits == "...
.text:0000000100036662                 mov     ebp, 10000h
.text:0000000100036667                 mov     ecx, ebp        ; unsigned ⤦
    ↳ int
.text:0000000100036669                 call    ?Log@@YAXIPEBGZZ ; Log(uint,⤦
    ↳ ushort const *,...)
.text:000000010003666E                 mov     r8d, 52         ; ---
.text:0000000100036674                 lea     rdx, aCardstackCreat_0 ; "⤦
    ↳ CardStack::CreateDeck()::uiNumCards == "...
.text:000000010003667B                 mov     ecx, ebp        ; unsigned ⤦
    ↳ int
.text:000000010003667D                 call    ?Log@@YAXIPEBGZZ ; Log(uint,⤦
    ↳ ushort const *,...)
.text:0000000100036682                 xor     edi, edi

.text:0000000100036684 loc_100036684:                          ; CODE XREF:⤦
    ↳  SolitaireGame::InitialDeal(void)+C0
.text:0000000100036684                 mov     eax, 4EC4EC4Fh
.text:0000000100036689                 mul     edi
.text:000000010003668B                 mov     r8d, edx
.text:000000010003668E                 shr     r8d, 4          ; unsigned ⤦
```

```
    ↳ int
.text:0000000100036692                  mov     eax, r8d
.text:0000000100036695                  imul    eax, 52         ; ---
.text:0000000100036698                  mov     edx, edi
.text:000000010003669A                  sub     edx, eax        ; unsigned ↙
    ↳ int
.text:000000010003669C                  mov     rcx, [rbx+128h] ; this
.text:00000001000366A3                  call    ?↙
    ↳ CreateCard@CardTable@@IEAAPEAVCard@@II@Z ; CardTable::CreateCard(uint↙
    ↳ ,uint)
.text:00000001000366A8                  mov     rdx, rax        ; struct ↙
    ↳ Card *
.text:00000001000366AB                  mov     rcx, rbx        ; this
.text:00000001000366AE                  call    ?↙
    ↳ Push@CardStack@@QEAAXPEAVCard@@@Z ; CardStack::Push(Card *)
.text:00000001000366B3                  inc     edi
.text:00000001000366B5                  cmp     edi, 52         ; ---
.text:00000001000366B8                  jb      short loc_100036684

.text:00000001000366BA                  xor     r8d, r8d        ; bool
.text:00000001000366BD                  xor     edx, edx        ; bool
.text:00000001000366BF                  mov     rcx, rbx        ; this
.text:00000001000366C2                  call    ?↙
    ↳ Arrange@CardStack@@QEAAX_N0@Z ; CardStack::Arrange(bool,bool)
.text:00000001000366C7                  mov     r13, [rsi+88h]
.text:00000001000366CE                  lea     rdx, aCardstackShuff ; "↙
    ↳ CardStack::Shuffle()"
.text:00000001000366D5                  mov     ecx, ebp        ; unsigned ↙
    ↳ int
.text:00000001000366D7                  call    ?Log@@YAXIPEBGZZ ; Log(uint,↙
    ↳ ushort const *,...)
.text:00000001000366DC                  and     [rsp+78h+var_40], 0
.text:00000001000366E1                  and     [rsp+78h+var_3C], 0
.text:00000001000366E6                  mov     [rsp+78h+var_38], 10h
.text:00000001000366EE                  xor     ebx, ebx
.text:00000001000366F0                  mov     [rsp+78h+var_30], rbx

...
```

De toutes façons, nous voyons clairement une boucle avec 52 itérations. Le corps de la boucle possède des appels à `CardTable()::CreateCard()` et `CardStack::Push()`.

La fonction `CardTable::CreateCard()` appelle finalement `Card::Init()` avec des valeurs dans l'intervalle 0..51, dans l'un de ses arguments. Ceci peut être vérifié facilement dans un débogueur.

Donc j'ai essayé de simplement changer le nombre 52 (0x34) en 51 (0x33) dans l'instruction `cmp edi, 52` en `0x1000366B5` et de le lancer. À première vue, rien ne s'est passé, mais j'ai remarqué qu'il était maintenant difficile de résoudre le jeu. J'ai passé presque une heure pour atteindre cette *position* :



Il manque l'as de cœur. Peut-être qu'en interne, cette carte a l'indice 51 (si les indices partent de zéro).

À un autre endroit, j'ai trouvé tous les noms des cartes. Peut-être que les noms sont utilisés pour aller chercher l'image de la carte dans les ressources?

```
.data:00000001000B6970 ?CARD_NAME@Card@@2PAPEBGA dq offset aTwoofclubs
.data:00000001000B6970                                         ; "↙
    ↳ TwoOfClubs"
.data:00000001000B6978                  dq offset aThreeofclubs ; "↙
    ↳ ThreeOfClubs"
.data:00000001000B6980                  dq offset aFourofclubs  ; "↙
    ↳ FourOfClubs"
.data:00000001000B6988                  dq offset aFiveofclubs  ; "↙
    ↳ FiveOfClubs"
.data:00000001000B6990                  dq offset aSixofclubs   ; "↙
    ↳ SixOfClubs"
.data:00000001000B6998                  dq offset aSevenofclubs ; "↙
    ↳ SevenOfClubs"
.data:00000001000B69A0                  dq offset aEightofclubs ; "↙
    ↳ EightOfClubs"
.data:00000001000B69A8                  dq offset aNineofclubs  ; "↙
    ↳ NineOfClubs"
.data:00000001000B69B0                  dq offset aTenofclubs   ; "↙
    ↳ TenOfClubs"
```

```
.data:00000001000B69B8                 dq offset aJackofclubs  ; "↙
    ↳ JackOfClubs"
.data:00000001000B69C0                 dq offset aQueenofclubs ; "↙
    ↳ QueenOfClubs"
.data:00000001000B69C8                 dq offset aKingofclubs  ; "↙
    ↳ KingOfClubs"
.data:00000001000B69D0                 dq offset aAceofclubs   ; "↙
    ↳ AceOfClubs"
.data:00000001000B69D8                 dq offset aTwoofdiamonds ; "↙
    ↳ TwoOfDiamonds"
.data:00000001000B69E0                 dq offset aThreeofdiamond ; "↙
    ↳ ThreeOfDiamonds"
.data:00000001000B69E8                 dq offset aFourofdiamonds ; "↙
    ↳ FourOfDiamonds"
.data:00000001000B69F0                 dq offset aFiveofdiamonds ; "↙
    ↳ FiveOfDiamonds"
.data:00000001000B69F8                 dq offset aSixofdiamonds ; "↙
    ↳ SixOfDiamonds"
.data:00000001000B6A00                 dq offset aSevenofdiamond ; "↙
    ↳ SevenOfDiamonds"
.data:00000001000B6A08                 dq offset aEightofdiamond ; "↙
    ↳ EightOfDiamonds"
.data:00000001000B6A10                 dq offset aNineofdiamonds ; "↙
    ↳ NineOfDiamonds"
.data:00000001000B6A18                 dq offset aTenofdiamonds ; "↙
    ↳ TenOfDiamonds"
.data:00000001000B6A20                 dq offset aJackofdiamonds ; "↙
    ↳ JackOfDiamonds"
.data:00000001000B6A28                 dq offset aQueenofdiamond ; "↙
    ↳ QueenOfDiamonds"
.data:00000001000B6A30                 dq offset aKingofdiamonds ; "↙
    ↳ KingOfDiamonds"
.data:00000001000B6A38                 dq offset aAceofdiamonds ; "↙
    ↳ AceOfDiamonds"
.data:00000001000B6A40                 dq offset aTwoofspades  ; "↙
    ↳ TwoOfSpades"
.data:00000001000B6A48                 dq offset aThreeofspades ; "↙
    ↳ ThreeOfSpades"
.data:00000001000B6A50                 dq offset aFourofspades ; "↙
    ↳ FourOfSpades"
.data:00000001000B6A58                 dq offset aFiveofspades ; "↙
    ↳ FiveOfSpades"
.data:00000001000B6A60                 dq offset aSixofspades  ; "↙
    ↳ SixOfSpades"
.data:00000001000B6A68                 dq offset aSevenofspades ; "↙
    ↳ SevenOfSpades"
.data:00000001000B6A70                 dq offset aEightofspades ; "↙
    ↳ EightOfSpades"
.data:00000001000B6A78                 dq offset aNineofspades ; "↙
    ↳ NineOfSpades"
.data:00000001000B6A80                 dq offset aTenofspades  ; "↙
    ↳ TenOfSpades"
.data:00000001000B6A88                 dq offset aJackofspades ; "↙
```

```
    ↳ JackOfSpades"
.data:00000001000B6A90                 dq offset aQueenofspades ; "↙
    ↳ QueenOfSpades"
.data:00000001000B6A98                 dq offset aKingofspades ; "↙
    ↳ KingOfSpades"
.data:00000001000B6AA0                 dq offset aAceofspades  ; "↙
    ↳ AceOfSpades"
.data:00000001000B6AA8                 dq offset aTwoofhearts  ; "↙
    ↳ TwoOfHearts"
.data:00000001000B6AB0                 dq offset aThreeofhearts ; "↙
    ↳ ThreeOfHearts"
.data:00000001000B6AB8                 dq offset aFourofhearts ; "↙
    ↳ FourOfHearts"
.data:00000001000B6AC0                 dq offset aFiveofhearts ; "↙
    ↳ FiveOfHearts"
.data:00000001000B6AC8                 dq offset aSixofhearts  ; "↙
    ↳ SixOfHearts"
.data:00000001000B6AD0                 dq offset aSevenofhearts ; "↙
    ↳ SevenOfHearts"
.data:00000001000B6AD8                 dq offset aEightofhearts ; "↙
    ↳ EightOfHearts"
.data:00000001000B6AE0                 dq offset aNineofhearts ; "↙
    ↳ NineOfHearts"
.data:00000001000B6AE8                 dq offset aTenofhearts  ; "↙
    ↳ TenOfHearts"
.data:00000001000B6AF0                 dq offset aJackofhearts ; "↙
    ↳ JackOfHearts"
.data:00000001000B6AF8                 dq offset aQueenofhearts ; "↙
    ↳ QueenOfHearts"
.data:00000001000B6B00                 dq offset aKingofhearts ; "↙
    ↳ KingOfHearts"
.data:00000001000B6B08                 dq offset aAceofhearts  ; "↙
    ↳ AceOfHearts"

.data:00000001000B6B10 ; public: static unsigned short const * near * Card↙
    ↳ ::CARD_HUMAN_NAME
.data:00000001000B6B10 ?CARD_HUMAN_NAME@Card@@2PAPEBGA dq offset ↙
    ↳ a54639Cardnames
.data:00000001000B6B10                                         ; "|54639|↙
    ↳ CardNames|Two Of Clubs"
.data:00000001000B6B18                 dq offset a64833Cardnames ; "|64833|↙
    ↳ CardNames|Three Of Clubs"
.data:00000001000B6B20                 dq offset a62984Cardnames ; "|62984|↙
    ↳ CardNames|Four Of Clubs"
.data:00000001000B6B28                 dq offset a65200Cardnames ; "|65200|↙
    ↳ CardNames|Five Of Clubs"
.data:00000001000B6B30                 dq offset a52967Cardnames ; "|52967|↙
    ↳ CardNames|Six Of Clubs"
.data:00000001000B6B38                 dq offset a42781Cardnames ; "|42781|↙
    ↳ CardNames|Seven Of Clubs"
.data:00000001000B6B40                 dq offset a49217Cardnames ; "|49217|↙
    ↳ CardNames|Eight Of Clubs"
.data:00000001000B6B48                 dq offset a44682Cardnames ; "|44682|↙
```

```
    ↳ CardNames|Nine Of Clubs"
.data:00000001000B6B50                 dq offset a51853Cardnames ; "|51853|↙
    ↳ CardNames|Ten Of Clubs"
.data:00000001000B6B58                 dq offset a46368Cardnames ; "|46368|↙
    ↳ CardNames|Jack Of Clubs"
.data:00000001000B6B60                 dq offset a61344Cardnames ; "|61344|↙
    ↳ CardNames|Queen Of Clubs"
.data:00000001000B6B68                 dq offset a65017Cardnames ; "|65017|↙
    ↳ CardNames|King Of Clubs"
.data:00000001000B6B70                 dq offset a57807Cardnames ; "|57807|↙
    ↳ CardNames|Ace Of Clubs"
.data:00000001000B6B78                 dq offset a48455Cardnames ; "|48455|↙
    ↳ CardNames|Two Of Diamonds"
.data:00000001000B6B80                 dq offset a44156Cardnames ; "|44156|↙
    ↳ CardNames|Three Of Diamonds"
.data:00000001000B6B88                 dq offset a51672Cardnames ; "|51672|↙
    ↳ CardNames|Four Of Diamonds"
.data:00000001000B6B90                 dq offset a45972Cardnames ; "|45972|↙
    ↳ CardNames|Five Of Diamonds"
.data:00000001000B6B98                 dq offset a47206Cardnames ; "|47206|↙
    ↳ CardNames|Six Of Diamonds"
.data:00000001000B6BA0                 dq offset a48399Cardnames ; "|48399|↙
    ↳ CardNames|Seven Of Diamonds"
.data:00000001000B6BA8                 dq offset a47847Cardnames ; "|47847|↙
    ↳ CardNames|Eight Of Diamonds"
.data:00000001000B6BB0                 dq offset a48606Cardnames ; "|48606|↙
    ↳ CardNames|Nine Of Diamonds"
.data:00000001000B6BB8                 dq offset a61278Cardnames ; "|61278|↙
    ↳ CardNames|Ten Of Diamonds"
.data:00000001000B6BC0                 dq offset a52038Cardnames ; "|52038|↙
    ↳ CardNames|Jack Of Diamonds"
.data:00000001000B6BC8                 dq offset a54643Cardnames ; "|54643|↙
    ↳ CardNames|Queen Of Diamonds"
.data:00000001000B6BD0                 dq offset a48902Cardnames ; "|48902|↙
    ↳ CardNames|King Of Diamonds"
.data:00000001000B6BD8                 dq offset a46672Cardnames ; "|46672|↙
    ↳ CardNames|Ace Of Diamonds"
.data:00000001000B6BE0                 dq offset a41049Cardnames ; "|41049|↙
    ↳ CardNames|Two Of Spades"
.data:00000001000B6BE8                 dq offset a49327Cardnames ; "|49327|↙
    ↳ CardNames|Three Of Spades"
.data:00000001000B6BF0                 dq offset a51933Cardnames ; "|51933|↙
    ↳ CardNames|Four Of Spades"
.data:00000001000B6BF8                 dq offset a42651Cardnames ; "|42651|↙
    ↳ CardNames|Five Of Spades"
.data:00000001000B6C00                 dq offset a65342Cardnames ; "|65342|↙
    ↳ CardNames|Six Of Spades"
.data:00000001000B6C08                 dq offset a53644Cardnames ; "|53644|↙
    ↳ CardNames|Seven Of Spades"
.data:00000001000B6C10                 dq offset a54466Cardnames ; "|54466|↙
    ↳ CardNames|Eight Of Spades"
.data:00000001000B6C18                 dq offset a56874Cardnames ; "|56874|↙
    ↳ CardNames|Nine Of Spades"
```

```
.data:00000001000B6C20                 dq offset a46756Cardnames ; "|46756|↙
    ↳ CardNames|Ten Of Spades"
.data:00000001000B6C28                 dq offset a62876Cardnames ; "|62876|↙
    ↳ CardNames|Jack Of Spades"
.data:00000001000B6C30                 dq offset a64633Cardnames ; "|64633|↙
    ↳ CardNames|Queen Of Spades"
.data:00000001000B6C38                 dq offset a46215Cardnames ; "|46215|↙
    ↳ CardNames|King Of Spades"
.data:00000001000B6C40                 dq offset a60450Cardnames ; "|60450|↙
    ↳ CardNames|Ace Of Spades"
.data:00000001000B6C48                 dq offset a51010Cardnames ; "|51010|↙
    ↳ CardNames|Two Of Hearts"
.data:00000001000B6C50                 dq offset a64948Cardnames ; "|64948|↙
    ↳ CardNames|Three Of Hearts"
.data:00000001000B6C58                 dq offset a43079Cardnames ; "|43079|↙
    ↳ CardNames|Four Of Hearts"
.data:00000001000B6C60                 dq offset a57131Cardnames ; "|57131|↙
    ↳ CardNames|Five Of Hearts"
.data:00000001000B6C68                 dq offset a58953Cardnames ; "|58953|↙
    ↳ CardNames|Six Of Hearts"
.data:00000001000B6C70                 dq offset a45105Cardnames ; "|45105|↙
    ↳ CardNames|Seven Of Hearts"
.data:00000001000B6C78                 dq offset a47775Cardnames ; "|47775|↙
    ↳ CardNames|Eight Of Hearts"
.data:00000001000B6C80                 dq offset a41825Cardnames ; "|41825|↙
    ↳ CardNames|Nine Of Hearts"
.data:00000001000B6C88                 dq offset a41501Cardnames ; "|41501|↙
    ↳ CardNames|Ten Of Hearts"
.data:00000001000B6C90                 dq offset a47108Cardnames ; "|47108|↙
    ↳ CardNames|Jack Of Hearts"
.data:00000001000B6C98                 dq offset a55659Cardnames ; "|55659|↙
    ↳ CardNames|Queen Of Hearts"
.data:00000001000B6CA0                 dq offset a44572Cardnames ; "|44572|↙
    ↳ CardNames|King Of Hearts"
.data:00000001000B6CA8                 dq offset a44183Cardnames ; "|44183|↙
    ↳ CardNames|Ace Of Hearts"
```

Si vous voulez faire ceci à quelqu'un, assurez-vous que sa santé mentale est stable.

À part les noms de fonction dans le fichier PDB, il y a de nombreux appels à la fonction Log() qui peuvent grandement aider, car le jeu Solitaire signale ce qu'il est en train de faire en ce moment.

Devoir: essayer de *supprimer* quelques cartes ou le deux de trèfle. Et que se passe-t-il si nous échangeons les noms des cartes dans les tableaux de chaînes?

J'ai aussi essayé de passer des nombres comme 0, 0..50 à `Card:Init()` (pour avoir 2 zéro dans une liste de 52 nombres). Ainsi, j'ai vu deux cartes *deux de trèfle* à un moment, mais le Solitaire avait un comportement erratique.

Ceci est le Solitaire de Windows 7 modifié: Solitaire51.exe.

8.6.2 53 cartes

Maintenant, regardons la première partie de la boucle:

```
.text:0000000100036684 loc_100036684:                          ; CODE XREF:⤦
    ↳  SolitaireGame::InitialDeal(void)+↓C0j
.text:0000000100036684                 mov     eax, 4EC4EC4Fh
.text:0000000100036689                 mul     edi
.text:000000010003668B                 mov     r8d, edx
.text:000000010003668E                 shr     r8d, 4          ; unsigned ⤦
    ↳ int
.text:0000000100036692                 mov     eax, r8d
.text:0000000100036695                 imul    eax, 52
.text:0000000100036698                 mov     edx, edi
.text:000000010003669A                 sub     edx, eax        ; unsigned ⤦
    ↳ int
.text:000000010003669C                 mov     rcx, [rbx+128h] ; this
.text:00000001000366A3                 call    ?⤦
    ↳ CreateCard@CardTable@@IEAAPEAVCard@@II@Z ; CardTable::CreateCard(uint⤦
    ↳ ,uint)
.text:00000001000366A8                 mov     rdx, rax        ; struct ⤦
    ↳ Card *
.text:00000001000366AB                 mov     rcx, rbx        ; this
.text:00000001000366AE                 call    ?⤦
    ↳ Push@CardStack@@QEAAXPEAVCard@@@Z ; CardStack::Push(Card *)
.text:00000001000366B3                 inc     edi
.text:00000001000366B5                 cmp     edi, 52
.text:00000001000366B8                 jb      short loc_100036684
```

Qu'est-ce que cette multiplication par 4EC4EC4Fh? Il s'agit sûrement de la division par la multiplication. Et voici ce qu'Hex-Rays en dit:

```
  v5 = 0;
  do
  {
    v6 = CardTable::CreateCard(v4[37], v5 % 0x34, v5 / 0x34);
    CardStack::Push((CardStack *)v4, v6);
    ++v5;
  }
  while ( v5 < 0x34 );
```

D'une certaine façon, la fonction `CreateCard()` prend deux arguments: l'itérateur divisé par 52 et le reste de l'opération de division. Difficile de dire pourquoi ils ont fait ainsi. Le Solitaire ne peut pas permettre plus de 52 cartes, donc le dernier argument est absurde, il vaut toujours zéro.

Mais une fois que j'ai modifié l'instruction `cmp edi, 52` en 0x1000366B5 par `cmp edi, 53`, j'ai trouvé qu'il y avait maintenant 53 cartes. La dernière est le *deux de trèfle*, car il s'agit de la carte numérotée 0.

Lors de la dernière itération, 0x52 est divisé par 0x52, le reste est zéro, donc la carte d'indice 0 est ajoutée deux fois.

Que c'est frustrant, il y a deux *deux de trèfle* :

Ceci est le Solitaire de Windows 7 modifié: Solitaire53.exe.

8.7 Blague FreeCell (Windows 7)

Ceci est une blague que j'ai fait une fois pour mes collègues qui jouaient trop au solitaire FreeCell. Pouvons-nous forcer FreeCell à jouer la même partie à chaque fois? Comme, voyez-vous, dans le film “Groundhog Day” ?

(J'écris ceci en novembre 2019. Il semble qu'IDA ne puisse obtenir les PDBs depuis les serveurs de Microsoft. Peut-être que Windows 7 n'est plus supporté? En tout cas, je ne peux pas obtenir les noms de fonction...)

8.7.1 Partie I

Donc, j'ai chargé FreeCell.exe dans IDA et trouvé qu'à la fois rand(), srand() et time() sont importées depuis msvcrt.dll. time() est en effet utilisée comme valeur d'initialisation pour srand() :

```
.text:01029612                  sub_1029612      proc near                  ; CODE
    XREF: sub_102615C+149
```

```
.text:01029612                                                   ;
    sub_1029DA6+67
.text:01029612 8B FF                           mov     edi, edi
.text:01029614 56                              push    esi
.text:01029615 57                              push    edi
.text:01029616 6A 00                           push    0               ; Time
.text:01029618 8B F9                           mov     edi, ecx
.text:0102961A FF 15 80 16 00+                 call    ds:time
.text:01029620 50                              push    eax             ; Seed
.text:01029621 FF 15 84 16 00+                 call    ds:srand
.text:01029627 8B 35 AC 16 00+                 mov     esi, ds:rand
.text:0102962D 59                              pop     ecx
.text:0102962E 59                              pop     ecx
.text:0102962F FF D6                           call    esi ; rand
.text:01029631 FF D6                           call    esi ; rand
.text:01029633
.text:01029633                loc_1029633:                             ; CODE
    XREF: sub_1029612+26
.text:01029633                                                   ;
    sub_1029612+2D
.text:01029633 FF D6                           call    esi ; rand
.text:01029635 83 F8 01                        cmp     eax, 1
.text:01029638 7C F9                           jl      short loc_1029633
.text:0102963A 3D 40 42 0F 00                  cmp     eax, 1000000
.text:0102963F 7F F2                           jg      short loc_1029633
.text:01029641 6A 01                           push    1
.text:01029643 50                              push    eax
.text:01029644 8B CF                           mov     ecx, edi
.text:01029646 E8 2D F8 FF FF                  call    sub_1028E78
.text:0102964B 5F                              pop     edi
.text:0102964C 5E                              pop     esi
.text:0102964D C3                              retn
.text:0102964D                sub_1029612      endp
```

“In the morning you will send for a hansom, desiring your man to take neither the first nor the second which may present itself.” (The Memoirs of Sherlock Holmes, par Arthur Conan Doyle[12])

Il y a un autre appel a la parie time() et srand(), mais mon tracer a montré que ceci est notre point d’intérêt:

```
tracer.exe -l:FreeCell.exe bpf=msvcrt.dll!time bpf=msvcrt.dll!srand,args:1

...

TID=5340|(0) msvcrt.dll!time() (called from FreeCell.exe!BASE+0x29620 (0↙
    ↘ x209620))
TID=5340|(0) msvcrt.dll!time() -> 0x5ddb68aa
TID=5340|(1) msvcrt.dll!srand(0x5ddb68aa) (called from FreeCell.exe!BASE+0↙
    ↘ x29627 (0x209627))
TID=5340|(1) msvcrt.dll!srand() -> 0x5507e0
TID=5340|(1) msvcrt.dll!srand(0x399f) (called from FreeCell.exe!BASE+0↙
    ↘ x27d3a (0x207d3a))
```

[12] http://www.gutenberg.org/files/834/834-0.txt

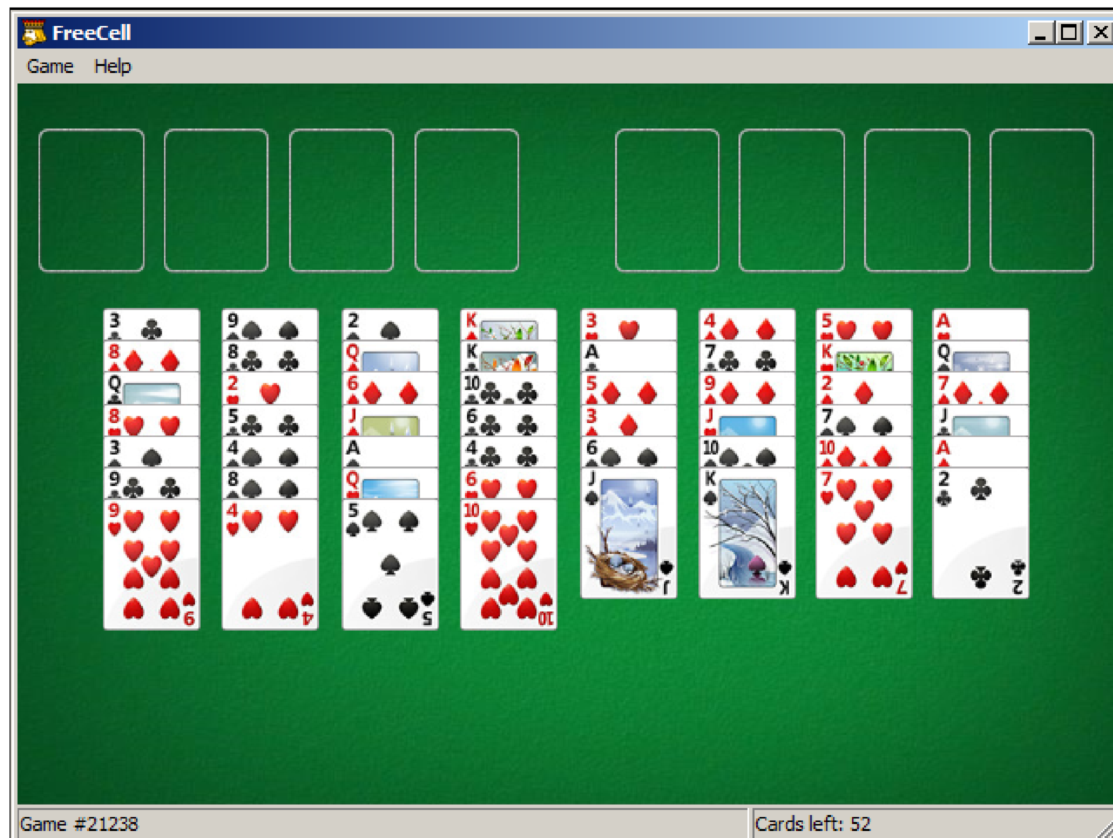
```
TID=5340|(1) msvcrt.dll!srand() -> 0x5507e0
```

Vous voyez, la fonction time() a renvoyé 0x5ddb68aa et la même valeur est utilisée comme un argument pour srand().

Essayons de forcer time() a toujours renvoyé 0:

```
tracer.exe -l:FreeCell.exe bpf=msvcrt.dll!time,rt:0 bpf=msvcrt.dll!srand,↙
    ↳ args:1

...

TID=2104|(0) msvcrt.dll!time() (called from FreeCell.exe!BASE+0x29620 (0↙
    ↳ xb19620))
TID=2104|(0) msvcrt.dll!time() -> 0x5ddb68f6
TID=2104|(0) Modifying EAX register to 0x0
TID=2104|(1) msvcrt.dll!srand(0x0) (called from FreeCell.exe!BASE+0x29627 ↙
    ↳ (0xb19627))
TID=2104|(1) msvcrt.dll!srand() -> 0x3707e0
TID=2104|(1) msvcrt.dll!srand(0x52f6) (called from FreeCell.exe!BASE+0↙
    ↳ x27d3a (0xb17d3a))
TID=2104|(1) msvcrt.dll!srand() -> 0x3707e0
```

Maintenant, je vois toujours le même jeu à chaque fois que je lance FreeCell en utilisant tracer :



Maintenant, comment modifier l'exécutable?

Nous voulons passer 0 comme argument à srand() en 0x01029620. Mais il y a une instruction sur un octet: PUSH EAX. Or PUSH 0 est une instruction sur deux octets. Comment la faire tenir?

Qui a-t-il dans les autres registres à ce moment? En utilisant tracer je les affiche tout:

```
tracer.exe -l:FreeCell.exe bpx=FreeCell.exe!0x01029620

...

TID=4448|(0) FreeCell.exe!0x1029620
EAX=0x5ddb6ac4 EBX=0x00000000 ECX=0x00000000 EDX=0x00000000
ESI=0x054732d0 EDI=0x054732d0 EBP=0x0020f2bc ESP=0x0020f298
EIP=0x00899620
FLAGS=PF ZF IF
TID=4448|(0) FreeCell.exe!0x1029620
EAX=0x5ddb6ac8 EBX=0x00000002 ECX=0x00000000 EDX=0x00000000
ESI=0xffffff11 EDI=0x054732d0 EBP=0x0020da78 ESP=0x0020d9d4
EIP=0x00899620
FLAGS=PF ZF IF
TID=4448|(0) FreeCell.exe!0x1029620
EAX=0x5ddb6aca EBX=0x00000002 ECX=0x00000000 EDX=0x00000000
```

```
ESI=0x7740c460 EDI=0x054732d0 EBP=0x0020da78 ESP=0x0020d9d4
EIP=0x00899620
FLAGS=PF ZF IF
...
```

Peu importe le nombre de fois que je redémaare le jeu, ECX et EDX semblent toujours contenir 0. Donc, j'ai modifié `PUSH EAX` à l'adress 0x01029620 en `PUSH EDX` (aussi une instruction sur 1 octet), et maintenant FreeCell montre toujours le même jeu au joueur.

Toutefois, d'autres options pourraient exister. En fait, nous n'avons pas besoin de passer 0 à srand(). Plutôt, nous voulons passer une *constante* à srand() pour que le jeu soit le même à chaque fois. Comme on peut le voir, la valeur d'EDI n'a pas changé. Peut-être que nous pourrions l'essayer aussi.

Maintenant une modification un peu plus difficile. Ouvrons FreeCell.exe dans Hiew:

```
Hiew: FreeCell.exe
    C:\tmp\FreeCell.exe
.01029612: 8BFF                          mov         edi,edi
.01029614: 56                            push        esi
.01029615: 57                            push        edi
.01029616: 6A00                          push        0
.01029618: 8BF9                          mov         edi,ecx
.0102961A: FF1580160001                  call        time
.01029620: 50                            push        eax
.01029621: FF1584160001                  call        srand
.01029627: 8B35AC160001                  mov         esi,rand --□1
.0102962D: 59                            pop         ecx
.0102962E: 59                            pop         ecx
.0102962F: FFD6                          call        esi
.01029631: FFD6                          call        esi
.01029633: FFD6                         2call        esi
```

Nous n'avons pas des place pour remplacer l'instruction d'un octet `PUSH EAX` avec celle sur deux octets `PUSH 0`. Et nous ne pouvons pas juste remplir `CALL ds:time` avec des NOPs, car il y a un FIXUP (adresse de la fonction time() dans msvcrt.dll). (Hiew a marqué ces 4 octets en gris.) Donc, voici ce que je fais: modifier les 2 premiers octets en EB 04. Ceci est un JMP pour contourner les 4 octets FIXUP.

```
Hiew: FreeCell.exe
    C:\tmp\FreeCell.exe
.01029612: 8BFF                    mov        edi,edi
.01029614: 56                      push       esi
.01029615: 57                      push       edi
.01029616: 6A00                    push       0
.01029618: 8BF9                    mov        edi,ecx
.0102961A: EB04                    jmps       .001029620 --⊡1
.0102961C: 801600                  adc        b,[esi],0
.0102961F: 0190FF158416            add        [eax][0168415FF],edx
.01029625: 0001                    add        [ecx],al
.01029627: 8B35AC160001            mov        esi,rand --⊡2
.0102962D: 59                      pop        ecx
.0102962E: 90                      nop
.0102962F: FFD6                    call       esi
```

Puis, je remplace PUSH EAX avec NOP. Ainsi, srand() aura son argument du PUSH 0 au-dessus. Aussi, je modifie une des POP ECX en NOP, car j'ai supprimé un PUSH.

```
Hiew: FreeCell.exe
    C:\tmp\FreeCell.exe
.01029620: 90                      nop
.01029621: FF1584160001            call       srand
.01029627: 8B35AC160001            mov        esi,rand --⊡1
.0102962D: 59                      pop        ecx
.0102962E: 90                      nop
.0102962F: FFD6                    call       esi
.01029631: FFD6                    call       esi
.01029633: FFD6                   2call       esi
.01029635: 83F801                  cmp        eax,1
```

Maintenant, le chargeur de Windows écrita le FIXUP de 4 octets en 0x0102961C, mais ça m'est égal: l'adresse de time() ne sera plus utilisée.

8.7.2 Partie II: casser le sous-menu *Select Game*

L'utilisateur peut toujorus choisir des jeux différents dans le menu. Voyons si srand() est toujours appelée. J'essaye d'entrer 1/2/3 dans la boite de dialoue "Select Game":

```
tracer.exe -l:FreeCell.exe bpf=msvcrt.dll!srand,args:1

...

TID=4936|(0) msvcrt.dll!srand(0x5ddb6df9) (called from FreeCell.exe!BASE+0↙
    ↳ x29627 (0xb49627))
TID=4936|(0) msvcrt.dll!srand() -> 0x5907e0
TID=4936|(0) msvcrt.dll!srand(0x2b40) (called from FreeCell.exe!BASE+0↙
    ↳ x27d3a (0xb47d3a))
TID=4936|(0) msvcrt.dll!srand() -> 0x5907e0
```

```
TID=4936|(0) msvcrt.dll!srand(0x1) (called from FreeCell.exe!BASE+0x27d3a ↙
    ↳ (0xb47d3a))
TID=4936|(0) msvcrt.dll!srand() -> 0x5907e0
TID=4936|(0) msvcrt.dll!srand(0x2) (called from FreeCell.exe!BASE+0x27d3a ↙
    ↳ (0xb47d3a))
TID=4936|(0) msvcrt.dll!srand() -> 0x5907e0
TID=4936|(0) msvcrt.dll!srand(0x3) (called from FreeCell.exe!BASE+0x27d3a ↙
    ↳ (0xb47d3a))
TID=4936|(0) msvcrt.dll!srand() -> 0x5907e0
```

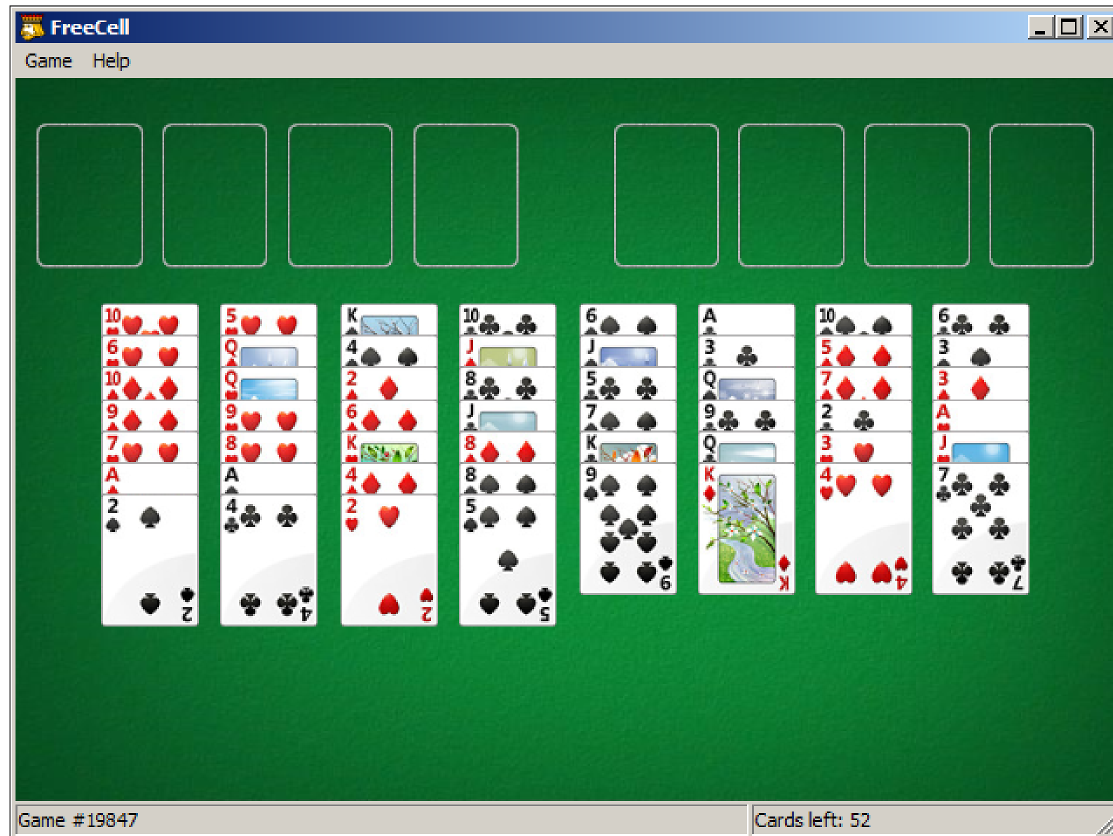
Oui, le nombre qu'entre l'utilisateur est simplement un argument pour srand(). Où est-elle appelée?

```
.text:01027CBA                loc_1027CBA:                          ; ↙
    ↳ CODE XREF: sub\_1027AC6+179
.text:01027CBA 83 FF FC                         cmp     edi, 0FFFFFFFCh
.text:01027CBD 75 74                            jnz     short loc_1027D33

...

.text:01027D33                loc_1027D33:                          ; ↙
    ↳ CODE XREF: sub\_1027AC6+1F7
.text:01027D33 57                               push    edi         ; ↙
    ↳ Seed
.text:01027D34 FF 15 84 16 00+                  call    ds:srand
.text:01027D3A 59                               pop     ecx
.text:01027D3B 6A 34                            push    34h
.text:01027D3D 5B                               pop     ebx
.text:01027D3E 33 C0                            xor     eax, eax
```

Je n'ai pas pu modifier `PUSH EDI` d'un octet en `PUSH 0` de deux octets. Mais je vois qu'il y seulement un unique saut à `loc_1027D33` dans ce qui précède.

Je modifie `CMP EDI, ...` en `XOR EDI, EDI`, en complètant le 3ème octet avec NOP. Je modifie aussi JNZ en JMP, afin que le saut se produise toujours.

Maintenant FreeCell ignore le nombre entré par l'utilisateur, mais soudain, il y a le même jeu au début:



Il semble que le code que nous avons modifié dans la partie I est relié d'une certaine façon à du code après 0x01027CBD, qui s'exécute si EDI==0xFFFFFFFC. De toutes façons, notre but est atteint — le jeu est toujours le même au début, et l'utilisateur ne peut pas en choisir un autre avec le menu.

8.8 Dongles

J'ai occasionnellement effectué des remplacements logiciel de dongle de protection de copie, ou «émulateur de dongle» et voici quelques exemples de comment ça s'est produit.

À propos du cas avec Rocket et Z3, qui n'est pas présent ici, vous pouvez le lire là: `http://yurichev.com/tmp/SAT_SMT_DRAFT.pdf`.

8.8.1 Exemple #1: MacOS Classic et PowerPC

Il y a ici un exemple de programme pour MacOS Classic[13], pour PowerPC. La société qui a développé le logiciel a disparu il y a longtemps, donc le client (légal) avait peur d'un problème matériel sur le dongle.

[13] pre-UNIX MacOS

Lorsqu'on le lançait sans le dongle connecté, une boite de dialogue avec le message "Invalid Security Device" apparaissait.

Par chance, cette chaîne de texte pût facilement être trouvée dans le fichier exécutable binaire.

Prétendons que nous ne sommes pas très familier, à la fois avec le Mac OS Classic et le PowerPC, mais essayons tout de même.

IDA ouvre le fichier exécutable sans problème, indique son type comme "PEF (Mac OS or Be OS executable)" (en effet, c'est un format de fichier Mac OS Classic standard).

En cherchant la chaîne de texte avec le message d'erreur, nous trouvons ce morceau de code:

```
...

seg000:000C87FC 38 60 00 01   li      %r3, 1
seg000:000C8800 48 03 93 41   bl      check1
seg000:000C8804 60 00 00 00   nop
seg000:000C8808 54 60 06 3F   clrlwi. %r0, %r3, 24
seg000:000C880C 40 82 00 40   bne     OK
seg000:000C8810 80 62 9F D8   lwz     %r3, TC_aInvalidSecurityDevice

...
```

Oui, ceci est du code PowerPC.

Le CPU est un RISC 32-bit très typique des années 1990s.

Chaque instruction occupe 4 octets (tout comme MIPS et ARM) et les noms ressemblent quelque peu aux noms des instructions MIPS.

check1() est une fonction donc nous allons donner le nom plu tard. BL est l'instruction *Branch Link*, e.g., destinée à appeler des sous-programmes.

Le point crucial est l'instruction BNE qui saute si la vérification du dongle de protection passe ou pas si une erreur se produit: alors l'adresse de la chaîne de texte est chargée dans le registre r3 pour la passer à la routine de la boite de dialogue subséquente.

Dans [Steve Zucker, SunSoft and Kari Karhi, IBM, *SYSTEM V APPLICATION BINARY INTERFACE: PowerPC Processor Supplement*, (1995)][14]nous trouvons que le registre r3 es utilisé pour la valeur de retour (et r4, dans le cas de valeurs 64-bit).

Une autre instruction inconnue est CLRLWI. Dans [*PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, (2000)][15]nous apprenons que cette instruction effectue la mise à zéro et le chargement. Dans notre cas, elle efface les 24 bits haut de la valeur dans r3 et la met dans r0, donc elle est analogue à MOVZX en x86 (1.23.1 on page 265), mais elle met aussi les flags, donc BNE peut ensuite les tester.

Jetons un œil à la fonction check1() :

[14]Aussi disponible en http://yurichev.com/mirrors/PowerPC/elfspec_ppc.pdf
[15]Aussi disponible en http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf

```
seg000:00101B40             check1: # CODE XREF: seg000:00063E7Cp
seg000:00101B40                     # sub_64070+160p ...
seg000:00101B40
seg000:00101B40             .set arg_8,  8
seg000:00101B40
seg000:00101B40 7C 08 02 A6         mflr    %r0
seg000:00101B44 90 01 00 08         stw     %r0, arg_8(%sp)
seg000:00101B48 94 21 FF C0         stwu    %sp, -0x40(%sp)
seg000:00101B4C 48 01 6B 39         bl      check2
seg000:00101B50 60 00 00 00         nop
seg000:00101B54 80 01 00 48         lwz     %r0, 0x40+arg_8(%sp)
seg000:00101B58 38 21 00 40         addi    %sp, %sp, 0x40
seg000:00101B5C 7C 08 03 A6         mtlr    %r0
seg000:00101B60 4E 80 00 20         blr
seg000:00101B60             # End of function check1
```

Comme on peut le voir dans IDA, cette fonction est appelée depuis de nombreux points du programme, mais seule la valeur du registre r3 est testée après chaque appel.

Tout ce que fait cette fonction est d'appeler une autre fonction, donc c'est une fonction thunk : il y a un prologue et un épilogue de fonction, mais le registre r3 n'est pas touché, donc checkl() renvoie ce que check2() renvoie.

BLR[16] semble être le retour de la fonction, mais vu comment IDA dispose la fonction, nous ne devons probablement pas nous en occuper.

Puisque c'est un RISC typique, il semble que les sous-programmes soient appelés en utilisant un registre de lien, tout comme en ARM.

La fonction check2() est plus complexe:

```
seg000:00118684             check2: # CODE XREF: check1+Cp
seg000:00118684
seg000:00118684             .set var_18, -0x18
seg000:00118684             .set var_C, -0xC
seg000:00118684             .set var_8, -8
seg000:00118684             .set var_4, -4
seg000:00118684             .set arg_8,  8
seg000:00118684
seg000:00118684 93 E1 FF FC   stw     %r31, var_4(%sp)
seg000:00118688 7C 08 02 A6   mflr    %r0
seg000:0011868C 83 E2 95 A8   lwz     %r31, off_1485E8 # dword_24B704
seg000:00118690               .using dword_24B704, %r31
seg000:00118690 93 C1 FF F8   stw     %r30, var_8(%sp)
seg000:00118694 93 A1 FF F4   stw     %r29, var_C(%sp)
seg000:00118698 7C 7D 1B 78   mr      %r29, %r3
seg000:0011869C 90 01 00 08   stw     %r0, arg_8(%sp)
seg000:001186A0 54 60 06 3E   clrlwi  %r0, %r3, 24
seg000:001186A4 28 00 00 01   cmplwi  %r0, 1
seg000:001186A8 94 21 FF B0   stwu    %sp, -0x50(%sp)
seg000:001186AC 40 82 00 0C   bne     loc_1186B8
```

[16](PowerPC) Branch to Link Register

```
seg000:001186B0 38 60 00 01   li      %r3, 1
seg000:001186B4 48 00 00 6C   b       exit
seg000:001186B8
seg000:001186B8            loc_1186B8: # CODE XREF: check2+28j
seg000:001186B8 48 00 03 D5   bl      sub_118A8C
seg000:001186BC 60 00 00 00   nop
seg000:001186C0 3B C0 00 00   li      %r30, 0
seg000:001186C4
seg000:001186C4            skip:       # CODE XREF: check2+94j
seg000:001186C4 57 C0 06 3F   clrlwi. %r0, %r30, 24
seg000:001186C8 41 82 00 18   beq     loc_1186E0
seg000:001186CC 38 61 00 38   addi    %r3, %sp, 0x50+var_18
seg000:001186D0 80 9F 00 00   lwz     %r4, dword_24B704
seg000:001186D4 48 00 C0 55   bl      .RBEFINDNEXT
seg000:001186D8 60 00 00 00   nop
seg000:001186DC 48 00 00 1C   b       loc_1186F8
seg000:001186E0
seg000:001186E0            loc_1186E0: # CODE XREF: check2+44j
seg000:001186E0 80 BF 00 00   lwz     %r5, dword_24B704
seg000:001186E4 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:001186E8 38 60 08 C2   li      %r3, 0x1234
seg000:001186EC 48 00 BF 99   bl      .RBEFINDFIRST
seg000:001186F0 60 00 00 00   nop
seg000:001186F4 3B C0 00 01   li      %r30, 1
seg000:001186F8
seg000:001186F8            loc_1186F8: # CODE XREF: check2+58j
seg000:001186F8 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:001186FC 41 82 00 0C   beq     must_jump
seg000:00118700 38 60 00 00   li      %r3, 0          # error
seg000:00118704 48 00 00 1C   b       exit
seg000:00118708
seg000:00118708            must_jump:  # CODE XREF: check2+78j
seg000:00118708 7F A3 EB 78   mr      %r3, %r29
seg000:0011870C 48 00 00 31   bl      check3
seg000:00118710 60 00 00 00   nop
seg000:00118714 54 60 06 3F   clrlwi. %r0, %r3, 24
seg000:00118718 41 82 FF AC   beq     skip
seg000:0011871C 38 60 00 01   li      %r3, 1
seg000:00118720
seg000:00118720            exit:       # CODE XREF: check2+30j
seg000:00118720                        # check2+80j
seg000:00118720 80 01 00 58   lwz     %r0, 0x50+arg_8(%sp)
seg000:00118724 38 21 00 50   addi    %sp, %sp, 0x50
seg000:00118728 83 E1 FF FC   lwz     %r31, var_4(%sp)
seg000:0011872C 7C 08 03 A6   mtlr    %r0
seg000:00118730 83 C1 FF F8   lwz     %r30, var_8(%sp)
seg000:00118734 83 A1 FF F4   lwz     %r29, var_C(%sp)
seg000:00118738 4E 80 00 20   blr
seg000:00118738            # End of function check2
```

Nous sommes encore chanceux: quelques noms de foncions ont été laissés dans l'exécutable (section de symboles de débogage?
difficile à dire puisque nous ne sommes pas très familier de ce format de fichier,

peut-être est-ce une sorte d'export PE? (6.5.2)),
comme .RBEFINDNEXT() et .RBEFINDFIRST().

Enfin ces fonctions appellent d'autres fonctions avec des noms comme .GetNextDeviceViaUSB(), .USBSendPKT(), donc elles travaillent clairement avec un dispositif USB.

Il y a même une fonction appelée .GetNextEve3Device()—qui semble familière, il y avait un dongle Sentinel Eve3 pour le port ADB (présent sue les MACs) dans les 1990s.

Regardons d'abord comment le registre r3 est mis avant le retour, en ignorant tout le reste.

Nous savons qu'une «bonne» valeur de r3 doit être non-nulle, r3 à zéro conduit à l'exécution de la partie avec un message d'erreur dans une boite de dialogue.

Il y a deux instructions `li %r3, 1` présentes dans la fonction et une `li %r3, 0` (*Load Immediate*, i.e., charger une valeur dans un registre). La première instruction est en 0x001186B0—et franchement, il est difficile de dire ce qu'elle signifie.

Ce que l'on voit ensuite, toutefois, est plus facile à comprendre: .RBEFINDFIRST() est appelé: si elle échoue, 0 est écrit dans r3 et nous sautons à *exit*, autrement une autre fonction est appelée(check3())—si elle échoue aussi, .RBEFINDNEXT() est appelée, probablement afin de chercher un autre dispositif USB.

N.B.: `clrlwi. %r0, %r3, 16` est analogue à ce que nous avons déjà vu, mais elle éfface 16 bits, i.e.,
.RBEFINDFIRST() renvoie probablement une valeur 16-bit.

B (signifie *branch*) saut inconditionnel.

BEQ est l'instruction inverse de BNE.

Regardons check3() :

```
seg000:0011873C             check3: # CODE XREF: check2+88p
seg000:0011873C
seg000:0011873C             .set var_18, -0x18
seg000:0011873C             .set var_C, -0xC
seg000:0011873C             .set var_8, -8
seg000:0011873C             .set var_4, -4
seg000:0011873C             .set arg_8,  8
seg000:0011873C
seg000:0011873C 93 E1 FF FC   stw     %r31, var_4(%sp)
seg000:00118740 7C 08 02 A6   mflr    %r0
seg000:00118744 38 A0 00 00   li      %r5, 0
seg000:00118748 93 C1 FF F8   stw     %r30, var_8(%sp)
seg000:0011874C 83 C2 95 A8   lwz     %r30, off_1485E8 # dword_24B704
seg000:00118750               .using dword_24B704, %r30
seg000:00118750 93 A1 FF F4   stw     %r29, var_C(%sp)
seg000:00118754 3B A3 00 00   addi    %r29, %r3, 0
seg000:00118758 38 60 00 00   li      %r3, 0
seg000:0011875C 90 01 00 08   stw     %r0, arg_8(%sp)
seg000:00118760 94 21 FF B0   stwu    %sp, -0x50(%sp)
seg000:00118764 80 DE 00 00   lwz     %r6, dword_24B704
seg000:00118768 38 81 00 38   addi    %r4, %sp, 0x50+var_18
```

```
seg000:0011876C 48 00 C0 5D   bl      .RBEREAD
seg000:00118770 60 00 00 00   nop
seg000:00118774 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118778 41 82 00 0C   beq     loc_118784
seg000:0011877C 38 60 00 00   li      %r3, 0
seg000:00118780 48 00 02 F0   b       exit
seg000:00118784
seg000:00118784             loc_118784: # CODE XREF: check3+3Cj
seg000:00118784 A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118788 28 00 04 B2   cmplwi  %r0, 0x1100
seg000:0011878C 41 82 00 0C   beq     loc_118798
seg000:00118790 38 60 00 00   li      %r3, 0
seg000:00118794 48 00 02 DC   b       exit
seg000:00118798
seg000:00118798             loc_118798: # CODE XREF: check3+50j
seg000:00118798 80 DE 00 00   lwz     %r6, dword_24B704
seg000:0011879C 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:001187A0 38 60 00 01   li      %r3, 1
seg000:001187A4 38 A0 00 00   li      %r5, 0
seg000:001187A8 48 00 C0 21   bl      .RBEREAD
seg000:001187AC 60 00 00 00   nop
seg000:001187B0 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:001187B4 41 82 00 0C   beq     loc_1187C0
seg000:001187B8 38 60 00 00   li      %r3, 0
seg000:001187BC 48 00 02 B4   b       exit
seg000:001187C0
seg000:001187C0             loc_1187C0: # CODE XREF: check3+78j
seg000:001187C0 A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:001187C4 28 00 06 4B   cmplwi  %r0, 0x09AB
seg000:001187C8 41 82 00 0C   beq     loc_1187D4
seg000:001187CC 38 60 00 00   li      %r3, 0
seg000:001187D0 48 00 02 A0   b       exit
seg000:001187D4
seg000:001187D4             loc_1187D4: # CODE XREF: check3+8Cj
seg000:001187D4 4B F9 F3 D9   bl      sub_B7BAC
seg000:001187D8 60 00 00 00   nop
seg000:001187DC 54 60 06 3E   clrlwi  %r0, %r3, 24
seg000:001187E0 2C 00 00 05   cmpwi   %r0, 5
seg000:001187E4 41 82 01 00   beq     loc_1188E4
seg000:001187E8 40 80 00 10   bge     loc_1187F8
seg000:001187EC 2C 00 00 04   cmpwi   %r0, 4
seg000:001187F0 40 80 00 58   bge     loc_118848
seg000:001187F4 48 00 01 8C   b       loc_118980
seg000:001187F8
seg000:001187F8             loc_1187F8: # CODE XREF: check3+ACj
seg000:001187F8 2C 00 00 0B   cmpwi   %r0, 0xB
seg000:001187FC 41 82 00 08   beq     loc_118804
seg000:00118800 48 00 01 80   b       loc_118980
seg000:00118804
seg000:00118804             loc_118804: # CODE XREF: check3+C0j
seg000:00118804 80 DE 00 00   lwz     %r6, dword_24B704
seg000:00118808 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:0011880C 38 60 00 08   li      %r3, 8
```

```
seg000:00118810 38 A0 00 00   li      %r5, 0
seg000:00118814 48 00 BF B5   bl      .RBEREAD
seg000:00118818 60 00 00 00   nop
seg000:0011881C 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118820 41 82 00 0C   beq     loc_11882C
seg000:00118824 38 60 00 00   li      %r3, 0
seg000:00118828 48 00 02 48   b       exit
seg000:0011882C
seg000:0011882C              loc_11882C: # CODE XREF: check3+E4j
seg000:0011882C A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118830 28 00 11 30   cmplwi  %r0, 0xFEA0
seg000:00118834 41 82 00 0C   beq     loc_118840
seg000:00118838 38 60 00 00   li      %r3, 0
seg000:0011883C 48 00 02 34   b       exit
seg000:00118840
seg000:00118840              loc_118840: # CODE XREF: check3+F8j
seg000:00118840 38 60 00 01   li      %r3, 1
seg000:00118844 48 00 02 2C   b       exit
seg000:00118848
seg000:00118848              loc_118848: # CODE XREF: check3+B4j
seg000:00118848 80 DE 00 00   lwz     %r6, dword_24B704
seg000:0011884C 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:00118850 38 60 00 0A   li      %r3, 0xA
seg000:00118854 38 A0 00 00   li      %r5, 0
seg000:00118858 48 00 BF 71   bl      .RBEREAD
seg000:0011885C 60 00 00 00   nop
seg000:00118860 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118864 41 82 00 0C   beq     loc_118870
seg000:00118868 38 60 00 00   li      %r3, 0
seg000:0011886C 48 00 02 04   b       exit
seg000:00118870
seg000:00118870              loc_118870: # CODE XREF: check3+128j
seg000:00118870 A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118874 28 00 03 F3   cmplwi  %r0, 0xA6E1
seg000:00118878 41 82 00 0C   beq     loc_118884
seg000:0011887C 38 60 00 00   li      %r3, 0
seg000:00118880 48 00 01 F0   b       exit
seg000:00118884
seg000:00118884              loc_118884: # CODE XREF: check3+13Cj
seg000:00118884 57 BF 06 3E   clrlwi  %r31, %r29, 24
seg000:00118888 28 1F 00 02   cmplwi  %r31, 2
seg000:0011888C 40 82 00 0C   bne     loc_118898
seg000:00118890 38 60 00 01   li      %r3, 1
seg000:00118894 48 00 01 DC   b       exit
seg000:00118898
seg000:00118898              loc_118898: # CODE XREF: check3+150j
seg000:00118898 80 DE 00 00   lwz     %r6, dword_24B704
seg000:0011889C 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:001188A0 38 60 00 0B   li      %r3, 0xB
seg000:001188A4 38 A0 00 00   li      %r5, 0
seg000:001188A8 48 00 BF 21   bl      .RBEREAD
seg000:001188AC 60 00 00 00   nop
seg000:001188B0 54 60 04 3F   clrlwi. %r0, %r3, 16
```

```
seg000:001188B4 41 82 00 0C   beq     loc_1188C0
seg000:001188B8 38 60 00 00   li      %r3, 0
seg000:001188BC 48 00 01 B4   b       exit
seg000:001188C0
seg000:001188C0             loc_1188C0: # CODE XREF: check3+178j
seg000:001188C0 A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:001188C4 28 00 23 1C   cmplwi  %r0, 0x1C20
seg000:001188C8 41 82 00 0C   beq     loc_1188D4
seg000:001188CC 38 60 00 00   li      %r3, 0
seg000:001188D0 48 00 01 A0   b       exit
seg000:001188D4
seg000:001188D4             loc_1188D4: # CODE XREF: check3+18Cj
seg000:001188D4 28 1F 00 03   cmplwi  %r31, 3
seg000:001188D8 40 82 01 94   bne     error
seg000:001188DC 38 60 00 01   li      %r3, 1
seg000:001188E0 48 00 01 90   b       exit
seg000:001188E4
seg000:001188E4             loc_1188E4: # CODE XREF: check3+A8j
seg000:001188E4 80 DE 00 00   lwz     %r6, dword_24B704
seg000:001188E8 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:001188EC 38 60 00 0C   li      %r3, 0xC
seg000:001188F0 38 A0 00 00   li      %r5, 0
seg000:001188F4 48 00 BE D5   bl      .RBEREAD
seg000:001188F8 60 00 00 00   nop
seg000:001188FC 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118900 41 82 00 0C   beq     loc_11890C
seg000:00118904 38 60 00 00   li      %r3, 0
seg000:00118908 48 00 01 68   b       exit
seg000:0011890C
seg000:0011890C             loc_11890C: # CODE XREF: check3+1C4j
seg000:0011890C A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118910 28 00 1F 40   cmplwi  %r0, 0x40FF
seg000:00118914 41 82 00 0C   beq     loc_118920
seg000:00118918 38 60 00 00   li      %r3, 0
seg000:0011891C 48 00 01 54   b       exit
seg000:00118920
seg000:00118920             loc_118920: # CODE XREF: check3+1D8j
seg000:00118920 57 BF 06 3E   clrlwi  %r31, %r29, 24
seg000:00118924 28 1F 00 02   cmplwi  %r31, 2
seg000:00118928 40 82 00 0C   bne     loc_118934
seg000:0011892C 38 60 00 01   li      %r3, 1
seg000:00118930 48 00 01 40   b       exit
seg000:00118934
seg000:00118934             loc_118934: # CODE XREF: check3+1ECj
seg000:00118934 80 DE 00 00   lwz     %r6, dword_24B704
seg000:00118938 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:0011893C 38 60 00 0D   li      %r3, 0xD
seg000:00118940 38 A0 00 00   li      %r5, 0
seg000:00118944 48 00 BE 85   bl      .RBEREAD
seg000:00118948 60 00 00 00   nop
seg000:0011894C 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118950 41 82 00 0C   beq     loc_11895C
seg000:00118954 38 60 00 00   li      %r3, 0
```

```
seg000:00118958 48 00 01 18   b       exit
seg000:0011895C
seg000:0011895C              loc_11895C: # CODE XREF: check3+214j
seg000:0011895C A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118960 28 00 07 CF   cmplwi  %r0, 0xFC7
seg000:00118964 41 82 00 0C   beq     loc_118970
seg000:00118968 38 60 00 00   li      %r3, 0
seg000:0011896C 48 00 01 04   b       exit
seg000:00118970
seg000:00118970              loc_118970: # CODE XREF: check3+228j
seg000:00118970 28 1F 00 03   cmplwi  %r31, 3
seg000:00118974 40 82 00 F8   bne     error
seg000:00118978 38 60 00 01   li      %r3, 1
seg000:0011897C 48 00 00 F4   b       exit
seg000:00118980
seg000:00118980              loc_118980: # CODE XREF: check3+B8j
seg000:00118980                            # check3+C4j
seg000:00118980 80 DE 00 00   lwz     %r6, dword_24B704
seg000:00118984 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:00118988 3B E0 00 00   li      %r31, 0
seg000:0011898C 38 60 00 04   li      %r3, 4
seg000:00118990 38 A0 00 00   li      %r5, 0
seg000:00118994 48 00 BE 35   bl      .RBEREAD
seg000:00118998 60 00 00 00   nop
seg000:0011899C 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:001189A0 41 82 00 0C   beq     loc_1189AC
seg000:001189A4 38 60 00 00   li      %r3, 0
seg000:001189A8 48 00 00 C8   b       exit
seg000:001189AC
seg000:001189AC              loc_1189AC: # CODE XREF: check3+264j
seg000:001189AC A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:001189B0 28 00 1D 6A   cmplwi  %r0, 0xAED0
seg000:001189B4 40 82 00 0C   bne     loc_1189C0
seg000:001189B8 3B E0 00 01   li      %r31, 1
seg000:001189BC 48 00 00 14   b       loc_1189D0
seg000:001189C0
seg000:001189C0              loc_1189C0: # CODE XREF: check3+278j
seg000:001189C0 28 00 18 28   cmplwi  %r0, 0x2818
seg000:001189C4 41 82 00 0C   beq     loc_1189D0
seg000:001189C8 38 60 00 00   li      %r3, 0
seg000:001189CC 48 00 00 A4   b       exit
seg000:001189D0
seg000:001189D0              loc_1189D0: # CODE XREF: check3+280j
seg000:001189D0                            # check3+288j
seg000:001189D0 57 A0 06 3E   clrlwi  %r0, %r29, 24
seg000:001189D4 28 00 00 02   cmplwi  %r0, 2
seg000:001189D8 40 82 00 20   bne     loc_1189F8
seg000:001189DC 57 E0 06 3F   clrlwi. %r0, %r31, 24
seg000:001189E0 41 82 00 10   beq     good2
seg000:001189E4 48 00 4C 69   bl      sub_11D64C
seg000:001189E8 60 00 00 00   nop
seg000:001189EC 48 00 00 84   b       exit
seg000:001189F0
```

```
seg000:001189F0             good2:      # CODE XREF: check3+2A4j
seg000:001189F0 38 60 00 01   li      %r3, 1
seg000:001189F4 48 00 00 7C   b       exit
seg000:001189F8
seg000:001189F8             loc_1189F8: # CODE XREF: check3+29Cj
seg000:001189F8 80 DE 00 00   lwz     %r6, dword_24B704
seg000:001189FC 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:00118A00 38 60 00 05   li      %r3, 5
seg000:00118A04 38 A0 00 00   li      %r5, 0
seg000:00118A08 48 00 BD C1   bl      .RBEREAD
seg000:00118A0C 60 00 00 00   nop
seg000:00118A10 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118A14 41 82 00 0C   beq     loc_118A20
seg000:00118A18 38 60 00 00   li      %r3, 0
seg000:00118A1C 48 00 00 54   b       exit
seg000:00118A20
seg000:00118A20             loc_118A20: # CODE XREF: check3+2D8j
seg000:00118A20 A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118A24 28 00 11 D3   cmplwi  %r0, 0xD300
seg000:00118A28 40 82 00 0C   bne     loc_118A34
seg000:00118A2C 3B E0 00 01   li      %r31, 1
seg000:00118A30 48 00 00 14   b       good1
seg000:00118A34
seg000:00118A34             loc_118A34: # CODE XREF: check3+2ECj
seg000:00118A34 28 00 1A EB   cmplwi  %r0, 0xEBA1
seg000:00118A38 41 82 00 0C   beq     good1
seg000:00118A3C 38 60 00 00   li      %r3, 0
seg000:00118A40 48 00 00 30   b       exit
seg000:00118A44
seg000:00118A44             good1:      # CODE XREF: check3+2F4j
seg000:00118A44                           # check3+2FCj
seg000:00118A44 57 A0 06 3E   clrlwi  %r0, %r29, 24
seg000:00118A48 28 00 00 03   cmplwi  %r0, 3
seg000:00118A4C 40 82 00 20   bne     error
seg000:00118A50 57 E0 06 3F   clrlwi. %r0, %r31, 24
seg000:00118A54 41 82 00 10   beq     good
seg000:00118A58 48 00 4B F5   bl      sub_11D64C
seg000:00118A5C 60 00 00 00   nop
seg000:00118A60 48 00 00 10   b       exit
seg000:00118A64
seg000:00118A64             good:       # CODE XREF: check3+318j
seg000:00118A64 38 60 00 01   li      %r3, 1
seg000:00118A68 48 00 00 08   b       exit
seg000:00118A6C
seg000:00118A6C             error:      # CODE XREF: check3+19Cj
seg000:00118A6C                           # check3+238j ...
seg000:00118A6C 38 60 00 00   li      %r3, 0
seg000:00118A70
seg000:00118A70             exit:       # CODE XREF: check3+44j
seg000:00118A70                           # check3+58j ...
seg000:00118A70 80 01 00 58   lwz     %r0, 0x50+arg_8(%sp)
seg000:00118A74 38 21 00 50   addi    %sp, %sp, 0x50
seg000:00118A78 83 E1 FF FC   lwz     %r31, var_4(%sp)
```

```
seg000:00118A7C 7C 08 03 A6   mtlr    %r0
seg000:00118A80 83 C1 FF F8   lwz     %r30, var_8(%sp)
seg000:00118A84 83 A1 FF F4   lwz     %r29, var_C(%sp)
seg000:00118A88 4E 80 00 20   blr
seg000:00118A88              # End of function check3
```

Il y a de nombreux appels à `.RBEREAD()`.

Peut-être que la fonction renvoie une valeur du dongle, donc elles sont comparées ici avec des variables codées en dur en utilisant `CMPLWI`.

Nous voyons aussi que le registre r3 est aussi rempli avant chaque appel à `.RBEREAD()` avec une de ces valeurs: 0,1, 8, 0xA, 0xB, 0xC, 0xD, 4, 5. Probablement une adresse mémoire ou quelque chose comme ça?

Oui, en effet, en googlant ces noms de fonction il est facile de trouver le manuel du dongle Sentinel Eve3!

Peut-être n'avons nous pas besoin d'apprendre aucune autre instruction PowerPC: tout ce que fait cette fonction est seulement d'appeler `.RBEREAD()`, de comparer ses résultats avec les constantes et de renvoyer 1 si les comparaisons sont justes ou 0 autrement.

Ok, tout ce dont nous avons besoin est que la fonction `check1()` renvoie toujours 1 ou n'importe quelle valeur autre que zéro.

Mais puisque nous ne sommes pas très sûrs de nos connaissances des instructions PowerPC, nous allons être prudents: nous allons patcher le saut dans `check2()` en `0x001186FC` et en `0x00118718`.

En `0x001186FC` nous allons écrire les octets 0x48 et 0, convertissant ainsi l'instruction BEQ en un B (saut inconditionnel) : nous pouvons repérer son opcode sans même nous référer à [*PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, (2000)][17].

En `0x00118718` nous allons écrire 0x60 et 3 octets à zéro, la convertissant ainsi en une instruction NOP : Nous pouvons aussi repérer son opcode dans le code.

Et maintenant, tout fonctionne sans un dongle connecté.

En résumé, des petites modification telles que celles-ci peuvent être effectuées avec IDA et un minimum de connaissances en langage d'assemblage.

8.8.2 Exemple #2: SCO OpenServer

Un ancien logiciel pour SCO OpenServer de 1997 développé par une société qui a disparue depuis longtemps.

Il y a un driver de dongle special à installer dans le système, qui contient les chaînes de texte suivantes: «Copyright 1989, Rainbow Technologies, Inc., Irvine, CA» et «Sentinel Integrated Driver Ver. 3.0».

[17] Aussi disponible en `http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf`

Après l'installation du driver dans SCO OpenServer, ces fichiers apparaissent dans l'arborescence /dev:

```
/dev/rbsl8
/dev/rbsl9
/dev/rbsl10
```

Le programme renvoie une erreur lorsque le dongle n'est pas connecté, mais le message d'erreur n'est pas trouvé dans les exécutables.

Grâce à IDA, il est facile de charger l'exécutable COFF utilisé dans SCO OpenServer.

Essayons de trouver la chaîne «rbsl» et en effet, elle se trouve dans ce morceau de code:

```
.text:00022AB8        public SSQC
.text:00022AB8 SSQC   proc near ; CODE XREF: SSQ+7p
.text:00022AB8
.text:00022AB8 var_44 = byte ptr -44h
.text:00022AB8 var_29 = byte ptr -29h
.text:00022AB8 arg_0  = dword ptr  8
.text:00022AB8
.text:00022AB8        push    ebp
.text:00022AB9        mov     ebp, esp
.text:00022ABB        sub     esp, 44h
.text:00022ABE        push    edi
.text:00022ABF        mov     edi, offset unk_4035D0
.text:00022AC4        push    esi
.text:00022AC5        mov     esi, [ebp+arg_0]
.text:00022AC8        push    ebx
.text:00022AC9        push    esi
.text:00022ACA        call    strlen
.text:00022ACF        add     esp, 4
.text:00022AD2        cmp     eax, 2
.text:00022AD7        jnz     loc_22BA4
.text:00022ADD        inc     esi
.text:00022ADE        mov     al, [esi-1]
.text:00022AE1        movsx   eax, al
.text:00022AE4        cmp     eax, '3'
.text:00022AE9        jz      loc_22B84
.text:00022AEF        cmp     eax, '4'
.text:00022AF4        jz      loc_22B94
.text:00022AFA        cmp     eax, '5'
.text:00022AFF        jnz     short loc_22B6B
.text:00022B01        movsx   ebx, byte ptr [esi]
.text:00022B04        sub     ebx, '0'
.text:00022B07        mov     eax, 7
.text:00022B0C        add     eax, ebx
.text:00022B0E        push    eax
.text:00022B0F        lea     eax, [ebp+var_44]
.text:00022B12        push    offset aDevSlD  ; "/dev/sl%d"
.text:00022B17        push    eax
.text:00022B18        call    nl_sprintf
.text:00022B1D        push    0               ; int
```

```
.text:00022B1F        push    offset aDevRbsl8 ; char *
.text:00022B24        call    _access
.text:00022B29        add     esp, 14h
.text:00022B2C        cmp     eax, 0FFFFFFFFh
.text:00022B31        jz      short loc_22B48
.text:00022B33        lea     eax, [ebx+7]
.text:00022B36        push    eax
.text:00022B37        lea     eax, [ebp+var_44]
.text:00022B3A        push    offset aDevRbslD ; "/dev/rbsl%d"
.text:00022B3F        push    eax
.text:00022B40        call    nl_sprintf
.text:00022B45        add     esp, 0Ch
.text:00022B48
.text:00022B48 loc_22B48: ; CODE XREF: SSQC+79j
.text:00022B48        mov     edx, [edi]
.text:00022B4A        test    edx, edx
.text:00022B4C        jle     short loc_22B57
.text:00022B4E        push    edx             ; int
.text:00022B4F        call    _close
.text:00022B54        add     esp, 4
.text:00022B57
.text:00022B57 loc_22B57: ; CODE XREF: SSQC+94j
.text:00022B57        push    2               ; int
.text:00022B59        lea     eax, [ebp+var_44]
.text:00022B5C        push    eax             ; char *
.text:00022B5D        call    _open
.text:00022B62        add     esp, 8
.text:00022B65        test    eax, eax
.text:00022B67        mov     [edi], eax
.text:00022B69        jge     short loc_22B78
.text:00022B6B
.text:00022B6B loc_22B6B: ; CODE XREF: SSQC+47j
.text:00022B6B        mov     eax, 0FFFFFFFFh
.text:00022B70        pop     ebx
.text:00022B71        pop     esi
.text:00022B72        pop     edi
.text:00022B73        mov     esp, ebp
.text:00022B75        pop     ebp
.text:00022B76        retn
.text:00022B78
.text:00022B78 loc_22B78: ; CODE XREF: SSQC+B1j
.text:00022B78        pop     ebx
.text:00022B79        pop     esi
.text:00022B7A        pop     edi
.text:00022B7B        xor     eax, eax
.text:00022B7D        mov     esp, ebp
.text:00022B7F        pop     ebp
.text:00022B80        retn
.text:00022B84
.text:00022B84 loc_22B84: ; CODE XREF: SSQC+31j
.text:00022B84        mov     al, [esi]
.text:00022B86        pop     ebx
.text:00022B87        pop     esi
```

```
.text:00022B88        pop     edi
.text:00022B89        mov     ds:byte_407224, al
.text:00022B8E        mov     esp, ebp
.text:00022B90        xor     eax, eax
.text:00022B92        pop     ebp
.text:00022B93        retn
.text:00022B94
.text:00022B94 loc_22B94: ; CODE XREF: SSQC+3Cj
.text:00022B94        mov     al, [esi]
.text:00022B96        pop     ebx
.text:00022B97        pop     esi
.text:00022B98        pop     edi
.text:00022B99        mov     ds:byte_407225, al
.text:00022B9E        mov     esp, ebp
.text:00022BA0        xor     eax, eax
.text:00022BA2        pop     ebp
.text:00022BA3        retn
.text:00022BA4
.text:00022BA4 loc_22BA4: ; CODE XREF: SSQC+1Fj
.text:00022BA4        movsx   eax, ds:byte_407225
.text:00022BAB        push    esi
.text:00022BAC        push    eax
.text:00022BAD        movsx   eax, ds:byte_407224
.text:00022BB4        push    eax
.text:00022BB5        lea     eax, [ebp+var_44]
.text:00022BB8        push    offset a46CCS   ; "46%c%c%s"
.text:00022BBD        push    eax
.text:00022BBE        call    nl_sprintf
.text:00022BC3        lea     eax, [ebp+var_44]
.text:00022BC6        push    eax
.text:00022BC7        call    strlen
.text:00022BCC        add     esp, 18h
.text:00022BCF        cmp     eax, 1Bh
.text:00022BD4        jle     short loc_22BDA
.text:00022BD6        mov     [ebp+var_29], 0
.text:00022BDA
.text:00022BDA loc_22BDA: ; CODE XREF: SSQC+11Cj
.text:00022BDA        lea     eax, [ebp+var_44]
.text:00022BDD        push    eax
.text:00022BDE        call    strlen
.text:00022BE3        push    eax             ; unsigned int
.text:00022BE4        lea     eax, [ebp+var_44]
.text:00022BE7        push    eax             ; void *
.text:00022BE8        mov     eax, [edi]
.text:00022BEA        push    eax             ; int
.text:00022BEB        call    _write
.text:00022BF0        add     esp, 10h
.text:00022BF3        pop     ebx
.text:00022BF4        pop     esi
.text:00022BF5        pop     edi
.text:00022BF6        mov     esp, ebp
.text:00022BF8        pop     ebp
.text:00022BF9        retn
```

```
.text:00022BFA        db 0Eh dup(90h)
.text:00022BFA SSQC   endp
```

Oui, en effet, le programme doit communiquer d'une façon ou d'une autre avec le driver.

Le seul endroit où la fonction SSQC() est appelée est dans la fonction thunk :

```
.text:0000DBE8        public SSQ
.text:0000DBE8 SSQ    proc near ; CODE XREF: sys_info+A9p
.text:0000DBE8                  ; sys_info+CBp ...
.text:0000DBE8
.text:0000DBE8 arg_0  = dword ptr  8
.text:0000DBE8
.text:0000DBE8        push    ebp
.text:0000DBE9        mov     ebp, esp
.text:0000DBEB        mov     edx, [ebp+arg_0]
.text:0000DBEE        push    edx
.text:0000DBEF        call    SSQC
.text:0000DBF4        add     esp, 4
.text:0000DBF7        mov     esp, ebp
.text:0000DBF9        pop     ebp
.text:0000DBFA        retn
.text:0000DBFB SSQ    endp
```

SSQ() peut être appelé depuis au moins 2 fonctions.

L'une d'entre elles est:

```
.data:0040169C _51_52_53       dd offset aPressAnyKeyT_0 ;
    DATA XREF: init_sys+392r
.data:0040169C                                          ; sys_info+A1r
.data:0040169C                                          ; "PRESS ANY KEY TO
    CONTINUE: "
.data:004016A0                 dd offset a51            ; "51"
.data:004016A4                 dd offset a52            ; "52"
.data:004016A8                 dd offset a53            ; "53"

...

.data:004016B8 _3C_or_3E       dd offset a3c            ; DATA XREF:
    sys_info:loc_D67Br
.data:004016B8                                          ; "3C"
.data:004016BC                 dd offset a3e            ; "3E"

; these names we gave to the labels:
.data:004016C0 answers1        dd 6B05h                 ; DATA XREF:
    sys_info+E7r
.data:004016C4                 dd 3D87h
.data:004016C8 answers2        dd 3Ch                   ; DATA XREF:
    sys_info+F2r
.data:004016CC                 dd 832h
.data:004016D0 _C_and_B        db 0Ch                   ; DATA XREF:
    sys_info+BAr
.data:004016D0                                          ; sys_info:OKr
```

```
.data:004016D1 byte_4016D1     db 0Bh                ;
    DATA XREF: sys_info+FDr
.data:004016D2                 db    0

...

.text:0000D652                 xor     eax, eax
.text:0000D654                 mov     al, ds:ctl_port
.text:0000D659                 mov     ecx, _51_52_53[eax*4]
.text:0000D660                 push    ecx
.text:0000D661                 call    SSQ
.text:0000D666                 add     esp, 4
.text:0000D669                 cmp     eax, 0FFFFFFFFh
.text:0000D66E                 jz      short loc_D6D1
.text:0000D670                 xor     ebx, ebx
.text:0000D672                 mov     al, _C_and_B
.text:0000D677                 test    al, al
.text:0000D679                 jz      short loc_D6C0
.text:0000D67B
.text:0000D67B loc_D67B: ; CODE XREF: sys_info+106j
.text:0000D67B                 mov     eax, _3C_or_3E[ebx*4]
.text:0000D682                 push    eax
.text:0000D683                 call    SSQ
.text:0000D688                 push    offset a4g      ; "4G"
.text:0000D68D                 call    SSQ
.text:0000D692                 push    offset a0123456789 ; "0123456789"
.text:0000D697                 call    SSQ
.text:0000D69C                 add     esp, 0Ch
.text:0000D69F                 mov     edx, answers1[ebx*4]
.text:0000D6A6                 cmp     eax, edx
.text:0000D6A8                 jz      short OK
.text:0000D6AA                 mov     ecx, answers2[ebx*4]
.text:0000D6B1                 cmp     eax, ecx
.text:0000D6B3                 jz      short OK
.text:0000D6B5                 mov     al, byte_4016D1[ebx]
.text:0000D6BB                 inc     ebx
.text:0000D6BC                 test    al, al
.text:0000D6BE                 jnz     short loc_D67B
.text:0000D6C0
.text:0000D6C0 loc_D6C0: ; CODE XREF: sys_info+C1j
.text:0000D6C0                 inc     ds:ctl_port
.text:0000D6C6                 xor     eax, eax
.text:0000D6C8                 mov     al, ds:ctl_port
.text:0000D6CD                 cmp     eax, edi
.text:0000D6CF                 jle     short loc_D652
.text:0000D6D1
.text:0000D6D1 loc_D6D1: ; CODE XREF: sys_info+98j
.text:0000D6D1           ; sys_info+B6j
.text:0000D6D1                 mov     edx, [ebp+var_8]
.text:0000D6D4                 inc     edx
.text:0000D6D5                 mov     [ebp+var_8], edx
.text:0000D6D8                 cmp     edx, 3
.text:0000D6DB                 jle     loc_D641
.text:0000D6E1
```

```
.text:0000D6E1 loc_D6E1: ; CODE XREF: sys_info+16j
.text:0000D6E1           ; sys_info+51j ...
.text:0000D6E1                 pop     ebx
.text:0000D6E2                 pop     edi
.text:0000D6E3                 mov     esp, ebp
.text:0000D6E5                 pop     ebp
.text:0000D6E6                 retn
.text:0000D6E8 OK:       ; CODE XREF: sys_info+F0j
.text:0000D6E8           ; sys_info+FBj
.text:0000D6E8                 mov     al, _C_and_B[ebx]
.text:0000D6EE                 pop     ebx
.text:0000D6EF                 pop     edi
.text:0000D6F0                 mov     ds:ctl_model, al
.text:0000D6F5                 mov     esp, ebp
.text:0000D6F7                 pop     ebp
.text:0000D6F8                 retn
.text:0000D6F8 sys_info        endp
```

« 3C » et « 3E » semblent familiers: il y avait un dongle Sentinel Pro de Rainbow sans mémoire, fournissant seulement une fonction de crypto-hachage secrète.

Vous pouvez lire une courte description de la fonction de hachage dont il s'agit ici: 2.5 on page 589.

Mais retournons au programme.

Donc le programme peut seulement tester si un dongle est connecté ou s'il est absent.

Aucune autre information ne peut être écrite dans un tel dongle, puisqu'il n'a pas de mémoire. Les codes sur deux caractères sont des commandes (nous pouvons voir comment les commandes sont traitées dans la fonction `SSQC()`) et toutes les autres chaînes sont hachées dans le dongle, transformées en un nombre 16-bit. L'algorithme était secret, donc il n'était pas possible d'écrire un driver de remplacement ou de refaire un dongle matériel qui l'émulerait parfaitement.

Toutefois, il est toujours possible d'intercepter tous les accès au dongle et de trouver les constantes auxquelles les résultats de la fonction de hachage sont comparées.

Mais nous devons dire qu'il est possible de construire un schéma de logiciel de protection de copie robuste basé sur une fonction secrète de hachage cryptographique: il suffit qu'elle chiffre/déchiffre les fichiers de données utilisés par votre logiciel.

Mais retournons au code:

Les codes 51/52/53 sont utilisés pour choisir le port imprimante LPT. 3x/4x sont utilisés pour le choix de la «famille» (c'est ainsi que les dongles Sentinel Pro sont différenciés les uns des autres: plus d'un dongle peut être connecté sur un port LPT).

La seule chaîne passée à la fonction qui ne fasse pas 2 caractères est "0123456789".

Ensuite, le résultat est comparé à l'ensemble des résultats valides.

Si il est correct, 0xC ou 0xB est écrit dans la variable globale `ctl_model`.

Une autre chaîne de texte qui est passée est ”PRESS ANY KEY TO CONTINUE: ”, mais le résultat n’est pas testé. Difficile de dire pourquoi, probablement une erreur[18].

Voyons où la valeur de la variable globale `ctl_model` est utilisée.

Un tel endroit est:

```
.text:0000D708 prep_sys proc near ; CODE XREF: init_sys+46Ap
.text:0000D708
.text:0000D708 var_14   = dword ptr -14h
.text:0000D708 var_10   = byte ptr -10h
.text:0000D708 var_8    = dword ptr -8
.text:0000D708 var_2    = word ptr -2
.text:0000D708
.text:0000D708          push    ebp
.text:0000D709          mov     eax, ds:net_env
.text:0000D70E          mov     ebp, esp
.text:0000D710          sub     esp, 1Ch
.text:0000D713          test    eax, eax
.text:0000D715          jnz     short loc_D734
.text:0000D717          mov     al, ds:ctl_model
.text:0000D71C          test    al, al
.text:0000D71E          jnz     short loc_D77E
.text:0000D720          mov     [ebp+var_8], offset aIeCvulnvvOkgT_ ;
    "Ie-cvulnvV\\\bOKG]T_"
.text:0000D727          mov     edx, 7
.text:0000D72C          jmp     loc_D7E7

...

.text:0000D7E7 loc_D7E7: ; CODE XREF: prep_sys+24j
.text:0000D7E7           ; prep_sys+33j
.text:0000D7E7          push    edx
.text:0000D7E8          mov     edx, [ebp+var_8]
.text:0000D7EB          push    20h
.text:0000D7ED          push    edx
.text:0000D7EE          push    16h
.text:0000D7F0          call    err_warn
.text:0000D7F5          push    offset station_sem
.text:0000D7FA          call    ClosSem
.text:0000D7FF          call    startup_err
```

Si c’est 0, un message d’erreur chiffré est passé à une routine de déchiffrement et affiché.

La routine de déchiffrement de la chaîne semble être un simple xor :

```
.text:0000A43C err_warn        proc near              ; CODE XREF:
    prep_sys+E8p
.text:0000A43C                                        ; prep_sys2+2Fp ...
.text:0000A43C
.text:0000A43C var_55          = byte ptr -55h
.text:0000A43C var_54          = byte ptr -54h
```

[18]C’est un sentiment étrange de trouver un bug dans un logiciel aussi ancien.

```
.text:0000A43C arg_0          = dword ptr  8
.text:0000A43C arg_4          = dword ptr  0Ch
.text:0000A43C arg_8          = dword ptr  10h
.text:0000A43C arg_C          = dword ptr  14h
.text:0000A43C
.text:0000A43C                push    ebp
.text:0000A43D                mov     ebp, esp
.text:0000A43F                sub     esp, 54h
.text:0000A442                push    edi
.text:0000A443                mov     ecx, [ebp+arg_8]
.text:0000A446                xor     edi, edi
.text:0000A448                test    ecx, ecx
.text:0000A44A                push    esi
.text:0000A44B                jle     short loc_A466
.text:0000A44D                mov     esi, [ebp+arg_C] ; key
.text:0000A450                mov     edx, [ebp+arg_4] ; string
.text:0000A453
.text:0000A453 loc_A453:                               ; CODE XREF:
    err_warn+28j
.text:0000A453                xor     eax, eax
.text:0000A455                mov     al, [edx+edi]
.text:0000A458                xor     eax, esi
.text:0000A45A                add     esi, 3
.text:0000A45D                inc     edi
.text:0000A45E                cmp     edi, ecx
.text:0000A460                mov     [ebp+edi+var_55], al
.text:0000A464                jl      short loc_A453
.text:0000A466
.text:0000A466 loc_A466:                               ; CODE XREF:
    err_warn+Fj
.text:0000A466                mov     [ebp+edi+var_54], 0
.text:0000A46B                mov     eax, [ebp+arg_0]
.text:0000A46E                cmp     eax, 18h
.text:0000A473                jnz     short loc_A49C
.text:0000A475                lea     eax, [ebp+var_54]
.text:0000A478                push    eax
.text:0000A479                call    status_line
.text:0000A47E                add     esp, 4
.text:0000A481
.text:0000A481 loc_A481:                               ; CODE XREF:
    err_warn+72j
.text:0000A481                push    50h
.text:0000A483                push    0
.text:0000A485                lea     eax, [ebp+var_54]
.text:0000A488                push    eax
.text:0000A489                call    memset
.text:0000A48E                call    pcv_refresh
.text:0000A493                add     esp, 0Ch
.text:0000A496                pop     esi
.text:0000A497                pop     edi
.text:0000A498                mov     esp, ebp
.text:0000A49A                pop     ebp
.text:0000A49B                retn
.text:0000A49C
```

```
.text:0000A49C loc_A49C:                                ; CODE XREF:
    err_warn+37j
.text:0000A49C                 push    0
.text:0000A49E                 lea     eax, [ebp+var_54]
.text:0000A4A1                 mov     edx, [ebp+arg_0]
.text:0000A4A4                 push    edx
.text:0000A4A5                 push    eax
.text:0000A4A6                 call    pcv_lputs
.text:0000A4AB                 add     esp, 0Ch
.text:0000A4AE                 jmp     short loc_A481
.text:0000A4AE err_warn        endp
```

C’est pourquoi nous étions incapable de trouver le message d’erreur dans les fichiers exécutable, car ils sont chiffrés (ce qui est une pratique courante).

Un autre appel à la fonction de hachage SSQ() lui passe la chaîne «offln» et le résultat est comparé avec 0xFE81 et 0x12A9.

Si ils ne correspondent pas, ça se comporte comme une sorte de fonction timer() (peut-être en attente qu’un dongle mal connecté soit reconnecté et re-testé?) et ensuite déchiffre un autre message d’erreur à afficher.

```
.text:0000DA55 loc_DA55:                                ; CODE XREF:
    sync_sys+24Cj
.text:0000DA55                 push    offset aOffln    ; "offln"
.text:0000DA5A                 call    SSQ
.text:0000DA5F                 add     esp, 4
.text:0000DA62                 mov     dl, [ebx]
.text:0000DA64                 mov     esi, eax
.text:0000DA66                 cmp     dl, 0Bh
.text:0000DA69                 jnz     short loc_DA83
.text:0000DA6B                 cmp     esi, 0FE81h
.text:0000DA71                 jz      OK
.text:0000DA77                 cmp     esi, 0FFFFF8EFh
.text:0000DA7D                 jz      OK
.text:0000DA83
.text:0000DA83 loc_DA83:                                ; CODE XREF:
    sync_sys+201j
.text:0000DA83                 mov     cl, [ebx]
.text:0000DA85                 cmp     cl, 0Ch
.text:0000DA88                 jnz     short loc_DA9F
.text:0000DA8A                 cmp     esi, 12A9h
.text:0000DA90                 jz      OK
.text:0000DA96                 cmp     esi, 0FFFFFFF5h
.text:0000DA99                 jz      OK
.text:0000DA9F
.text:0000DA9F loc_DA9F:                                ; CODE XREF:
    sync_sys+220j
.text:0000DA9F                 mov     eax, [ebp+var_18]
.text:0000DAA2                 test    eax, eax
.text:0000DAA4                 jz      short loc_DAB0
.text:0000DAA6                 push    24h
.text:0000DAA8                 call    timer
.text:0000DAAD                 add     esp, 4
.text:0000DAB0
```

```
.text:0000DAB0 loc_DAB0:                                ; CODE XREF:
    sync_sys+23Cj
.text:0000DAB0                 inc     edi
.text:0000DAB1                 cmp     edi, 3
.text:0000DAB4                 jle     short loc_DA55
.text:0000DAB6                 mov     eax, ds:net_env
.text:0000DABB                 test    eax, eax
.text:0000DABD                 jz      short error

...

.text:0000DAF7 error:                                   ; CODE XREF:
    sync_sys+255j
.text:0000DAF7                                          ; sync_sys+274j ...
.text:0000DAF7                 mov     [ebp+var_8], offset ↙
    ↳ encrypted_error_message2
.text:0000DAFE                 mov     [ebp+var_C], 17h ; decrypting key
.text:0000DB05                 jmp     decrypt_end_print_message

...

; this name we gave to label:
.text:0000D9B6 decrypt_end_print_message:               ; CODE XREF:
    sync_sys+29Dj
.text:0000D9B6                                          ; sync_sys+2ABj
.text:0000D9B6                 mov     eax, [ebp+var_18]
.text:0000D9B9                 test    eax, eax
.text:0000D9BB                 jnz     short loc_D9FB
.text:0000D9BD                 mov     edx, [ebp+var_C] ; key
.text:0000D9C0                 mov     ecx, [ebp+var_8] ; string
.text:0000D9C3                 push    edx
.text:0000D9C4                 push    20h
.text:0000D9C6                 push    ecx
.text:0000D9C7                 push    18h
.text:0000D9C9                 call    err_warn
.text:0000D9CE                 push    0Fh
.text:0000D9D0                 push    190h
.text:0000D9D5                 call    sound
.text:0000D9DA                 mov     [ebp+var_18], 1
.text:0000D9E1                 add     esp, 18h
.text:0000D9E4                 call    pcv_kbhit
.text:0000D9E9                 test    eax, eax
.text:0000D9EB                 jz      short loc_D9FB

...

; this name we gave to label:
.data:00401736 encrypted_error_message2 db 74h, 72h, 78h, 43h, 48h, 6, 5Ah,↙
    ↳  49h, 4Ch, 2 dup(47h)
.data:00401736                 db 51h, 4Fh, 47h, 61h, 20h, 22h, 3Ch, 24h, ↙
    ↳ 33h, 36h, 76h
.data:00401736                 db 3Ah, 33h, 31h, 0Ch, 0, 0Bh, 1Fh, 7, 1Eh, ↙
    ↳ 1Ah
```

Passer outre le dongle est assez facile: il suffit de patcher tous les sauts après les instructions CMP pertinentes.

Une autre option est d'écrire notre propre driver SCO OpenServer, contenant une table de questions et de réponses, toutes celles qui sont présentent dans le programme.

Déchiffrer les messages d'erreur

À propos, nous pouvons aussi essayer de déchiffrer tous les messages d'erreurs. L'algorithme qui se trouve dans la fonction `err_warn()` est très simple, en effet:

Listing 8.5 : Decryption function

```
.text:0000A44D                 mov     esi, [ebp+arg_C] ; clef
.text:0000A450                 mov     edx, [ebp+arg_4] ; chaîne
.text:0000A453 loc_A453:
.text:0000A453                 xor     eax, eax
.text:0000A455                 mov     al, [edx+edi] ; charger l'octet
    chiffré
.text:0000A458                 xor     eax, esi      ; le déchiffré
.text:0000A45A                 add     esi, 3        ; changé la clef pour
    l'octet suivant
.text:0000A45D                 inc     edi
.text:0000A45E                 cmp     edi, ecx
.text:0000A460                 mov     [ebp+edi+var_55], al
.text:0000A464                 jl      short loc_A453
```

Comme on le voit, non seulement la chaîne est transmise à la fonction de déchiffrement mais aussi la clef:

```
.text:0000DAF7 error:                                  ; CODE XREF:
    sync_sys+255j
.text:0000DAF7                                         ; sync_sys+274j ...
.text:0000DAF7                 mov     [ebp+var_8], offset ↙
    ↳ encrypted_error_message2
.text:0000DAFE                 mov     [ebp+var_C], 17h ; decrypting key
.text:0000DB05                 jmp     decrypt_end_print_message

...

; this name we gave to label manually:
.text:0000D9B6 decrypt_end_print_message:              ; CODE XREF:
    sync_sys+29Dj
.text:0000D9B6                                         ; sync_sys+2ABj
.text:0000D9B6                 mov     eax, [ebp+var_18]
.text:0000D9B9                 test    eax, eax
.text:0000D9BB                 jnz     short loc_D9FB
.text:0000D9BD                 mov     edx, [ebp+var_C] ; key
.text:0000D9C0                 mov     ecx, [ebp+var_8] ; string
.text:0000D9C3                 push    edx
.text:0000D9C4                 push    20h
.text:0000D9C6                 push    ecx
.text:0000D9C7                 push    18h
.text:0000D9C9                 call    err_warn
```

L'algorithme est un simple xor : chaque octet est xoré avec la clef, mais la clef est incrémentée de 3 après le traitement de chaque octet.

Nous pouvons écrire un petit script Python pour vérifier notre hypothèse:

Listing 8.6 : Python 3.x

```
#!/usr/bin/python
import sys

msg=[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A]

key=0x17
tmp=key
for i in msg:
        sys.stdout.write ("%c" % (i^tmp))
        tmp=tmp+3
sys.stdout.flush()
```

Et il affiche: «check security device connection». Donc oui, ceci est le message déchiffré.

Il y a d'autres messages chiffrés, avec leur clef correspondante. Mais inutile de dire qu'il est possible de les déchiffrer sans leur clef. Premièrement, nous voyons que le clef est en fait un octet. C'est parce que l'instruction principale de déchiffrement (XOR) fonctionne au niveau de l'octet. La clef se trouve dans le registre ESI, mais seulement une partie de ESI d'un octet est utilisée. Ainsi, une clef pourrait être plus grande que 255, mais sa valeur est toujours arrondie.

En conséquence, nous pouvons simplement essayer de brute-forcer, en essayant toutes les clefs possible dans l'intervalle 0..255. Nous allons aussi écarter les messages comportants des caractères non-imprimable.

Listing 8.7 : Python 3.x

```
#!/usr/bin/python
import sys, curses.ascii

msgs=[
[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A],

[0x49, 0x65, 0x2D, 0x63, 0x76, 0x75, 0x6C, 0x6E, 0x76, 0x56, 0x5C,
8, 0x4F, 0x4B, 0x47, 0x5D, 0x54, 0x5F, 0x1D, 0x26, 0x2C, 0x33,
0x27, 0x28, 0x6F, 0x72, 0x75, 0x78, 0x7B, 0x7E, 0x41, 0x44],

[0x45, 0x61, 0x31, 0x67, 0x72, 0x79, 0x68, 0x52, 0x4A, 0x52, 0x50,
0x0C, 0x4B, 0x57, 0x43, 0x51, 0x58, 0x5B, 0x61, 0x37, 0x33, 0x2B,
0x39, 0x39, 0x3C, 0x38, 0x79, 0x3A, 0x30, 0x17, 0x0B, 0x0C],

[0x40, 0x64, 0x79, 0x75, 0x7F, 0x6F, 0x0, 0x4C, 0x40, 0x9, 0x4D, 0x5A,
0x46, 0x5D, 0x57, 0x49, 0x57, 0x3B, 0x21, 0x23, 0x6A, 0x38, 0x23,
```

```
0x36, 0x24, 0x2A, 0x7C, 0x3A, 0x1A, 0x6, 0x0D, 0x0E, 0x0A, 0x14,
0x10],

[0x72, 0x7C, 0x72, 0x79, 0x76, 0x0,
0x50, 0x43, 0x4A, 0x59, 0x5D, 0x5B, 0x41, 0x41, 0x1B, 0x5A,
0x24, 0x32, 0x2E, 0x29, 0x28, 0x70, 0x20, 0x22, 0x38, 0x28, 0x36,
0x0D, 0x0B, 0x48, 0x4B, 0x4E]]


def is_string_printable(s):
    return all(list(map(lambda x: curses.ascii.isprint(x), s)))

cnt=1
for msg in msgs:
        print ("message #%d" % cnt)
        for key in range(0,256):
                result=[]
                tmp=key
                for i in msg:
                        result.append (i^tmp)
                        tmp=tmp+3
                if is_string_printable (result):
                        print ("key=", key, "value=", "".join(list(map(chr,↙
    ↳  result))))
        cnt=cnt+1
```

Et nous obtenons:

Listing 8.8 : Results

```
message #1
key= 20 value= `eb^h%|``hudw|_af{n~f%ljmSbnwlpk
key= 21 value= ajc]i"}cawtgv{^bgto}g"millcmvkqh
key= 22 value= bkd\j#rbbvsfuz!cduh|d#bhomdlujni
key= 23 value= check security device connection
key= 24 value= lifbl!pd|tqhsx#ejwjbb!`nQofbshlo
message #2
key= 7 value= No security device found
key= 8 value= An#rbbvsVuz!cduhld#ghtme?!#!'!#!
message #3
key= 7 value= Bk<waoqNUpu$`yreoa\wpmpusj,bkIjh
key= 8 value= Mj?vfnrOjqv%gxqd``_vwlstlk/clHii
key= 9 value= Lm>ugasLkvw&fgpgag^uvcrwml.`mwhj
key= 10 value= Ol!td`tMhwx'efwfbf!tubuvnm!anvok
key= 11 value= No security device station found
key= 12 value= In#rjbvsnuz!{duhdd#r{`whho#gPtme
message #4
key= 14 value= Number of authorized users exceeded
key= 15 value= Ovlmdq!hg#`juknuhydk!vrbsp!Zy`dbefe
message #5
key= 17 value= check security device station
key= 18 value= `ijbh!td`tmhwx'efwfbf!tubuVnm!'!
```

Ici il y a un peu de déchet, mais nous pouvons rapidement trouver les messages en anglais.

À propos, puisque l'algorithme est un simple chiffrement xor, la même fonction peut être utilisée pour chiffrer les messages. Si besoin, nous pouvons chiffrer nos propres messages, et patcher le programme en les insérant.

8.8.3 Exemple #3: MS-DOS

Un autre très vieux logiciel pour MS-DOS de 1995, lui aussi développé par une société disparue depuis longtemps.

À l' ère pré-DOS extenders, presque tous les logiciels pour MS-DOS s'appuyaient sur sur des CPUs 8086 ou 80286, donc la code était massivement 16-bit.

Le code 16-bit est presque le même que celui déjà vu dans le livre, mais tous les registres sont 16-bit et il y a moins d'instructions disponibles.

L'environnement MS-DOS n'avait pas de système de drivers, et n'importe quel programme pouvait s'adresser au matériel via les ports, donc vous pouvez voir ici les instructions `OUT`/`IN`, qui sont présentes dans la plupart des drivers de nos jours (il est impossible d'accéder directement aux ports en mode utilisateur sur tous les OSes modernes).

Compte tenu de ceci, le programme MS-DOS qui fonctionne avec un dongle doit accéder le port imprimante LPT directement.

Donc nous devons simplement chercher des telles instructions. Et oui, elles y sont:

```
seg030:0034             out_port proc far ; CODE XREF: sent_pro+22p
seg030:0034                               ; sent_pro+2Ap ...
seg030:0034
seg030:0034             arg_0    = byte ptr  6
seg030:0034
seg030:0034 55                   push    bp
seg030:0035 8B EC                mov     bp, sp
seg030:0037 8B 16 7E E7          mov     dx, _out_port ; 0x378
seg030:003B 8A 46 06             mov     al, [bp+arg_0]
seg030:003E EE                   out     dx, al
seg030:003F 5D                   pop     bp
seg030:0040 CB                   retf
seg030:0040             out_port endp
```

(J'ai donné tous les noms de label dans cet exemple).

`out_port()` est référencé dans une seule fonction:

```
seg030:0041             sent_pro proc far ; CODE XREF: check_dongle+34p
seg030:0041
seg030:0041             var_3    = byte ptr -3
seg030:0041             var_2    = word ptr -2
seg030:0041             arg_0    = dword ptr  6
seg030:0041
seg030:0041 C8 04 00 00          enter   4, 0
seg030:0045 56                   push    si
```

```
seg030:0046 57                      push    di
seg030:0047 8B 16 82 E7             mov     dx, _in_port_1 ; 0x37A
seg030:004B EC                      in      al, dx
seg030:004C 8A D8                   mov     bl, al
seg030:004E 80 E3 FE                and     bl, 0FEh
seg030:0051 80 CB 04                or      bl, 4
seg030:0054 8A C3                   mov     al, bl
seg030:0056 88 46 FD                mov     [bp+var_3], al
seg030:0059 80 E3 1F                and     bl, 1Fh
seg030:005C 8A C3                   mov     al, bl
seg030:005E EE                      out     dx, al
seg030:005F 68 FF 00                push    0FFh
seg030:0062 0E                      push    cs
seg030:0063 E8 CE FF                call    near ptr out_port
seg030:0066 59                      pop     cx
seg030:0067 68 D3 00                push    0D3h
seg030:006A 0E                      push    cs
seg030:006B E8 C6 FF                call    near ptr out_port
seg030:006E 59                      pop     cx
seg030:006F 33 F6                   xor     si, si
seg030:0071 EB 01                   jmp     short loc_359D4
seg030:0073
seg030:0073                 loc_359D3: ; CODE XREF: sent_pro+37j
seg030:0073 46                      inc     si
seg030:0074
seg030:0074                 loc_359D4: ; CODE XREF: sent_pro+30j
seg030:0074 81 FE 96 00             cmp     si, 96h
seg030:0078 7C F9                   jl      short loc_359D3
seg030:007A 68 C3 00                push    0C3h
seg030:007D 0E                      push    cs
seg030:007E E8 B3 FF                call    near ptr out_port
seg030:0081 59                      pop     cx
seg030:0082 68 C7 00                push    0C7h
seg030:0085 0E                      push    cs
seg030:0086 E8 AB FF                call    near ptr out_port
seg030:0089 59                      pop     cx
seg030:008A 68 D3 00                push    0D3h
seg030:008D 0E                      push    cs
seg030:008E E8 A3 FF                call    near ptr out_port
seg030:0091 59                      pop     cx
seg030:0092 68 C3 00                push    0C3h
seg030:0095 0E                      push    cs
seg030:0096 E8 9B FF                call    near ptr out_port
seg030:0099 59                      pop     cx
seg030:009A 68 C7 00                push    0C7h
seg030:009D 0E                      push    cs
seg030:009E E8 93 FF                call    near ptr out_port
seg030:00A1 59                      pop     cx
seg030:00A2 68 D3 00                push    0D3h
seg030:00A5 0E                      push    cs
seg030:00A6 E8 8B FF                call    near ptr out_port
seg030:00A9 59                      pop     cx
seg030:00AA BF FF FF                mov     di, 0FFFFh
```

```
seg030:00AD EB 40                  jmp     short loc_35A4F
seg030:00AF
seg030:00AF               loc_35A0F: ; CODE XREF: sent_pro+BDj
seg030:00AF BE 04 00               mov     si, 4
seg030:00B2
seg030:00B2               loc_35A12: ; CODE XREF: sent_pro+ACj
seg030:00B2 D1 E7                  shl     di, 1
seg030:00B4 8B 16 80 E7            mov     dx, _in_port_2 ; 0x379
seg030:00B8 EC                     in      al, dx
seg030:00B9 A8 80                  test    al, 80h
seg030:00BB 75 03                  jnz     short loc_35A20
seg030:00BD 83 CF 01               or      di, 1
seg030:00C0
seg030:00C0               loc_35A20: ; CODE XREF: sent_pro+7Aj
seg030:00C0 F7 46 FE 08+           test    [bp+var_2], 8
seg030:00C5 74 05                  jz      short loc_35A2C
seg030:00C7 68 D7 00               push    0D7h ; '+'
seg030:00CA EB 0B                  jmp     short loc_35A37
seg030:00CC
seg030:00CC               loc_35A2C: ; CODE XREF: sent_pro+84j
seg030:00CC 68 C3 00               push    0C3h
seg030:00CF 0E                     push    cs
seg030:00D0 E8 61 FF               call    near ptr out_port
seg030:00D3 59                     pop     cx
seg030:00D4 68 C7 00               push    0C7h
seg030:00D7
seg030:00D7               loc_35A37: ; CODE XREF: sent_pro+89j
seg030:00D7 0E                     push    cs
seg030:00D8 E8 59 FF               call    near ptr out_port
seg030:00DB 59                     pop     cx
seg030:00DC 68 D3 00               push    0D3h
seg030:00DF 0E                     push    cs
seg030:00E0 E8 51 FF               call    near ptr out_port
seg030:00E3 59                     pop     cx
seg030:00E4 8B 46 FE               mov     ax, [bp+var_2]
seg030:00E7 D1 E0                  shl     ax, 1
seg030:00E9 89 46 FE               mov     [bp+var_2], ax
seg030:00EC 4E                     dec     si
seg030:00ED 75 C3                  jnz     short loc_35A12
seg030:00EF
seg030:00EF               loc_35A4F: ; CODE XREF: sent_pro+6Cj
seg030:00EF C4 5E 06               les     bx, [bp+arg_0]
seg030:00F2 FF 46 06               inc     word ptr [bp+arg_0]
seg030:00F5 26 8A 07               mov     al, es:[bx]
seg030:00F8 98                     cbw
seg030:00F9 89 46 FE               mov     [bp+var_2], ax
seg030:00FC 0B C0                  or      ax, ax
seg030:00FE 75 AF                  jnz     short loc_35A0F
seg030:0100 68 FF 00               push    0FFh
seg030:0103 0E                     push    cs
seg030:0104 E8 2D FF               call    near ptr out_port
seg030:0107 59                     pop     cx
seg030:0108 8B 16 82 E7            mov     dx, _in_port_1 ; 0x37A
```

```
seg030:010C EC                      in      al, dx
seg030:010D 8A C8                   mov     cl, al
seg030:010F 80 E1 5F                and     cl, 5Fh
seg030:0112 8A C1                   mov     al, cl
seg030:0114 EE                      out     dx, al
seg030:0115 EC                      in      al, dx
seg030:0116 8A C8                   mov     cl, al
seg030:0118 F6 C1 20                test    cl, 20h
seg030:011B 74 08                   jz      short loc_35A85
seg030:011D 8A 5E FD                mov     bl, [bp+var_3]
seg030:0120 80 E3 DF                and     bl, 0DFh
seg030:0123 EB 03                   jmp     short loc_35A88
seg030:0125
seg030:0125             loc_35A85: ; CODE XREF: sent_pro+DAj
seg030:0125 8A 5E FD                mov     bl, [bp+var_3]
seg030:0128
seg030:0128             loc_35A88: ; CODE XREF: sent_pro+E2j
seg030:0128 F6 C1 80                test    cl, 80h
seg030:012B 74 03                   jz      short loc_35A90
seg030:012D 80 E3 7F                and     bl, 7Fh
seg030:0130
seg030:0130             loc_35A90: ; CODE XREF: sent_pro+EAj
seg030:0130 8B 16 82 E7             mov     dx, _in_port_1 ; 0x37A
seg030:0134 8A C3                   mov     al, bl
seg030:0136 EE                      out     dx, al
seg030:0137 8B C7                   mov     ax, di
seg030:0139 5F                      pop     di
seg030:013A 5E                      pop     si
seg030:013B C9                      leave
seg030:013C CB                      retf
seg030:013C             sent_pro endp
```

Ceci est un «hashing» dongle Sentinel Pro, comme dans l'exemple précédent. C'est remarquable car des chaînes de texte sont passées ici, aussi, et des valeurs 16-bit sont renvoyées, puis comparées avec d'autres.

Donc, voici comment le Sentinel Pro est accédé via les ports.

L'adresse du port de sortie est en général 0x378, i.e., le port imprimante, où les données pour les vieilles imprimantes de l'ère pré-USB étaient passées.

Le port est uni-directionnel, car lorsqu'il a été développé, personne n'imaginait que quelqu'un aurait besoin de transférer de l'information depuis l'imprimante [19].

Le seul moyen d'obtenir des informations de l'imprimante est le registre d'état sur le port 0x379, qui contient des bits tels que «paper out», «ack», «busy»—ainsi l'imprimante peut signaler si elle est prête ou non et si elle a du papier.

Donc, le dongle renvoie de l'information dans l'un de ces bits, un bit à chaque itération.

[19]Si nous considérons seulement Centronics. Le standard IEEE 1284 suivant permet le transfert d'information depuis l'imprimante.

_in_port_2 contient l'adresse du mot d'état (0x379) et _in_port_1 contient le registre de contrôle d'adresse (0x37A).

Il semble que le dongle renvoie de l'information via le flag «busy» en seg030:00B9 : chaque bit est stocké dans le registre DI, qui est renvoyé à la fin de la fonction.

Que signifie tous ces octets envoyés sur le port de sortie? Difficile à dire. Peut-être des commandes pour le dongle.

Mais d'une manière générale, il n'est pas nécessaire de savoir: il est facile de résoudre notre tâche sans le savoir.

Voici la routine de vérification du dongle:

```
00000000 struct_0        struc ; (sizeof=0x1B)
00000000 field_0         db 25 dup(?)           ; string(C)
00000019 _A              dw ?
0000001B struct_0        ends

dseg:3CBC 61 63 72 75+_Q   struct_0 <'hello', 01122h>
dseg:3CBC 6E 00 00 00+       ; DATA XREF: check_dongle+2Eo

... skipped ...

dseg:3E00 63 6F 66 66+     struct_0 <'coffee', 7EB7h>
dseg:3E1B 64 6F 67 00+     struct_0 <'dog', 0FFADh>
dseg:3E36 63 61 74 00+     struct_0 <'cat', 0FF5Fh>
dseg:3E51 70 61 70 65+     struct_0 <'paper', 0FFDFh>
dseg:3E6C 63 6F 6B 65+     struct_0 <'coke', 0F568h>
dseg:3E87 63 6C 6F 63+     struct_0 <'clock', 55EAh>
dseg:3EA2 64 69 72 00+     struct_0 <'dir', 0FFAEh>
dseg:3EBD 63 6F 70 79+     struct_0 <'copy', 0F557h>


seg030:0145             check_dongle proc far ; CODE XREF: sub_3771D+3EP
seg030:0145
seg030:0145             var_6 = dword ptr -6
seg030:0145             var_2 = word ptr -2
seg030:0145
seg030:0145 C8 06 00 00       enter   6, 0
seg030:0149 56                push    si
seg030:014A 66 6A 00          push    large 0         ; newtime
seg030:014D 6A 00             push    0               ; cmd
seg030:014F 9A C1 18 00+      call    _biostime
seg030:0154 52                push    dx
seg030:0155 50                push    ax
seg030:0156 66 58             pop     eax
seg030:0158 83 C4 06          add     sp, 6
seg030:015B 66 89 46 FA       mov     [bp+var_6], eax
seg030:015F 66 3B 06 D8+      cmp     eax, _expiration
seg030:0164 7E 44             jle     short loc_35B0A
seg030:0166 6A 14             push    14h
seg030:0168 90                nop
seg030:0169 0E                push    cs
seg030:016A E8 52 00          call    near ptr get_rand
```

```
seg030:016D 59                  pop     cx
seg030:016E 8B F0               mov     si, ax
seg030:0170 6B C0 1B            imul    ax, 1Bh
seg030:0173 05 BC 3C            add     ax, offset _Q
seg030:0176 1E                  push    ds
seg030:0177 50                  push    ax
seg030:0178 0E                  push    cs
seg030:0179 E8 C5 FE            call    near ptr sent_pro
seg030:017C 83 C4 04            add     sp, 4
seg030:017F 89 46 FE            mov     [bp+var_2], ax
seg030:0182 8B C6               mov     ax, si
seg030:0184 6B C0 12            imul    ax, 18
seg030:0187 66 0F BF C0         movsx   eax, ax
seg030:018B 66 8B 56 FA         mov     edx, [bp+var_6]
seg030:018F 66 03 D0            add     edx, eax
seg030:0192 66 89 16 D8+        mov     _expiration, edx
seg030:0197 8B DE               mov     bx, si
seg030:0199 6B DB 1B            imul    bx, 27
seg030:019C 8B 87 D5 3C         mov     ax, _Q._A[bx]
seg030:01A0 3B 46 FE            cmp     ax, [bp+var_2]
seg030:01A3 74 05               jz      short loc_35B0A
seg030:01A5 B8 01 00            mov     ax, 1
seg030:01A8 EB 02               jmp     short loc_35B0C
seg030:01AA
seg030:01AA             loc_35B0A: ; CODE XREF: check_dongle+1Fj
seg030:01AA                        ; check_dongle+5Ej
seg030:01AA 33 C0               xor     ax, ax
seg030:01AC
seg030:01AC             loc_35B0C: ; CODE XREF: check_dongle+63j
seg030:01AC 5E                  pop     si
seg030:01AD C9                  leave
seg030:01AE CB                  retf
seg030:01AE             check_dongle endp
```

Puisque la routine peut être appelée très fréquemment, e.g., avant l'exécution de chaque fonctionnalité importante du logiciel, et accéder au ongle est en général lent (à cause du port de l'imprimante et aussi du MCU lent du dongle), ils ont probablement ajouté un moyen d'éviter le test du dongle, en vérifiant l'heure courante dans la fonction `biostime()`.

La fonction get_rand() utilise la fonction C standard:

```
seg030:01BF             get_rand proc far ; CODE XREF: check_dongle+25p
seg030:01BF
seg030:01BF             arg_0    = word ptr  6
seg030:01BF
seg030:01BF 55                     push    bp
seg030:01C0 8B EC                  mov     bp, sp
seg030:01C2 9A 3D 21 00+           call    _rand
seg030:01C7 66 0F BF C0            movsx   eax, ax
seg030:01CB 66 0F BF 56+           movsx   edx, [bp+arg_0]
seg030:01D0 66 0F AF C2            imul    eax, edx
seg030:01D4 66 BB 00 80+           mov     ebx, 8000h
seg030:01DA 66 99                  cdq
```

```
seg030:01DC 66 F7 FB             idiv    ebx
seg030:01DF 5D                   pop     bp
seg030:01E0 CB                   retf
seg030:01E0             get_rand endp
```

Donc la chaîne de texte est choisi au hasard, passé au dongle, et ensuite le résultat du hachage est comparé à la valeur correcte.

Les chaînes de texte semblent être construites aléatoirement aussi, lors du développement du logiciel.

Et voici comment la fonction principale de vérification du dongle est appelée:

```
seg033:087B 9A 45 01 96+   call    check_dongle
seg033:0880 0B C0          or      ax, ax
seg033:0882 74 62          jz      short OK
seg033:0884 83 3E 60 42+   cmp     word_620E0, 0
seg033:0889 75 5B          jnz     short OK
seg033:088B FF 06 60 42    inc     word_620E0
seg033:088F 1E             push    ds
seg033:0890 68 22 44       push    offset aTrupcRequiresA ;
    "This Software Requires a Software Lock\n"
seg033:0893 1E             push    ds
seg033:0894 68 60 E9       push    offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+   call    _strcpy
seg033:089C 83 C4 08       add     sp, 8
seg033:089F 1E             push    ds
seg033:08A0 68 42 44       push    offset aPleaseContactA ; "Please Contact
    ..."
seg033:08A3 1E             push    ds
seg033:08A4 68 60 E9       push    offset byte_6C7E0 ; dest
seg033:08A7 9A CD 64 00+   call    _strcat
```

Il est facile de contourner le dongle, il suffit de forcer la fonction `check_dongle()` à renvoyer toujours 0.

Par exemple, en insérant du code à son début:

```
mov ax,0
retf
```

Le lecteur attentif peut se rappeler que la fonction C `strcpy()` prend en général deux pointeurs dans ses arguments, mais nous voyons que 4 valeurs sont passées:

```
seg033:088F 1E                          push    ds
seg033:0890 68 22 44                    push    offset aTrupcRequiresA ;
    "This Software Requires a Software Lock\n"
seg033:0893 1E                          push    ds
seg033:0894 68 60 E9                    push    offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+                call    _strcpy
seg033:089C 83 C4 08                    add     sp, 8
```

Ceci est relatif au modèle de mémoire de MS-DOS. Vous pouvez en lire plus à ce sujet ici: 11.7 on page 1297.

Donc, comme vous pouvez le voir, `strcpy()` et toute autre fonction qui prend un/des pointeur(s) en argument travaille avec des paires 16-bit.

Retournons à notre exemple. DS est actuellement l'adresse du segment de données dans l'exécutable, où la chaîne de texte est stockée.

Dans la fonction `sent_pro()`, chaque octet de la chaîne est chargé en `seg030:00EF` : l'instruction LES charge simultanément la paire ES:BX depuis l'argument transmis.

Le `MOV` en `seg030:00F5` charge l'octet depuis la mémoire sur laquelle pointe la paire ES:BX.

8.9 Cas de base de données chiffrée #1

(Cette partie est apparue initialement dans mon blog le 26 août 2015. Discussion: https://news.ycombinator.com/item?id=10128684.)

8.9.1 Base64 et entropie

J'ai un fichier XML contenant des données chiffrées. Peut-être est-ce relatif à des commandes et/ou des information clients.


```
<?xml version = "1.0" encoding = "UTF-8"?>
<Orders>
        <Order>
                <OrderID>1</OrderID>
                <Data>yjmxhXUbhB/5MV45chPsXZWAJwIh1S0aD9lFn3XuJMSxJ3/E+UE3hsnH</Data>
        </Order>
        <Order>
                <OrderID>2</OrderID>
                <Data>0KGe/wnypFBjsy+U0C2P9fC5nDZP3XDZLMPCRaiBw9OjIk6Tu5U=</Data>
        </Order>
        <Order>
                <OrderID>3</OrderID>
                <Data>mqkXfdzvQKvEArdzh+zD9oETVGBFvcTBLs2ph1b5bYddExzp</Data>
        </Order>
        <Order>
                <OrderID>4</OrderID>
                <Data>FCx6JhIDqnESyT3HAepyE1BJ3cJd7wCk+APCRUeuNtZdpCvQ2MR/7kLXtfUHuA==</Data>
        </Order>
...
```


Le fichier est disponible ici.

Ce sont clairement des données encodées en base64, car toutes les chaînes consistent en des caractères Latin, chiffres, plus (+) et symbole slash (/). Il peut y avoir 1 ou 2

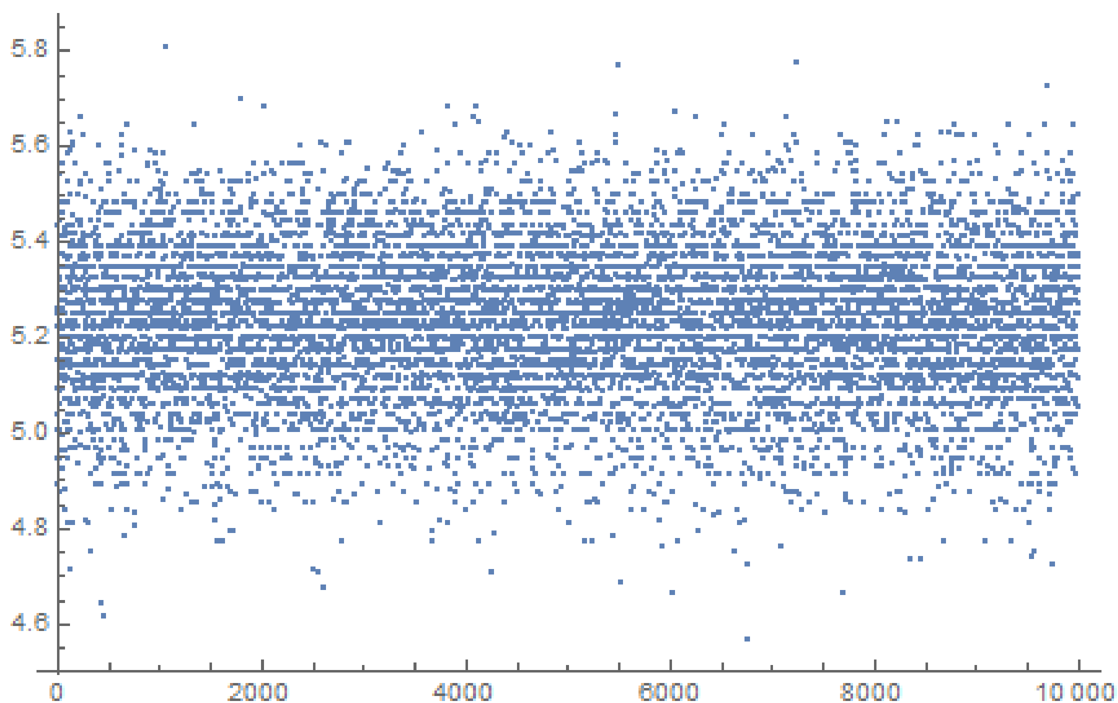
symboles de remplissage (=), mais ils ne se trouvent jamais au milieu d'une chaîne. Gardez à l'esprit ces propriétés du base64, il est très facile de les reconnaître.

Décodons les et calculons l'entropie (9.2 on page 1225) de ces blocs dans Wolfram Mathematica:

```
In[]:= ListOfBase64Strings =
  Map[First[#[[3]]] &, Cases[Import["encrypted.xml"], XMLElement["Data", _,↙
    ↳  _], Infinity]];

In[]:= BinaryStrings =
  Map[ImportString[#, {"Base64", "String"}] &, ListOfBase64Strings];

In[]:= Entropies = Map[N[Entropy[2, #]] &, BinaryStrings];

In[]:= Variance[Entropies]
Out[]= 0.0238614
```

La variance est basse. Cela signifie que l'entropie des valeurs ne sont pas très différentes les unes des autres. Ceci est visible sur le graphique:

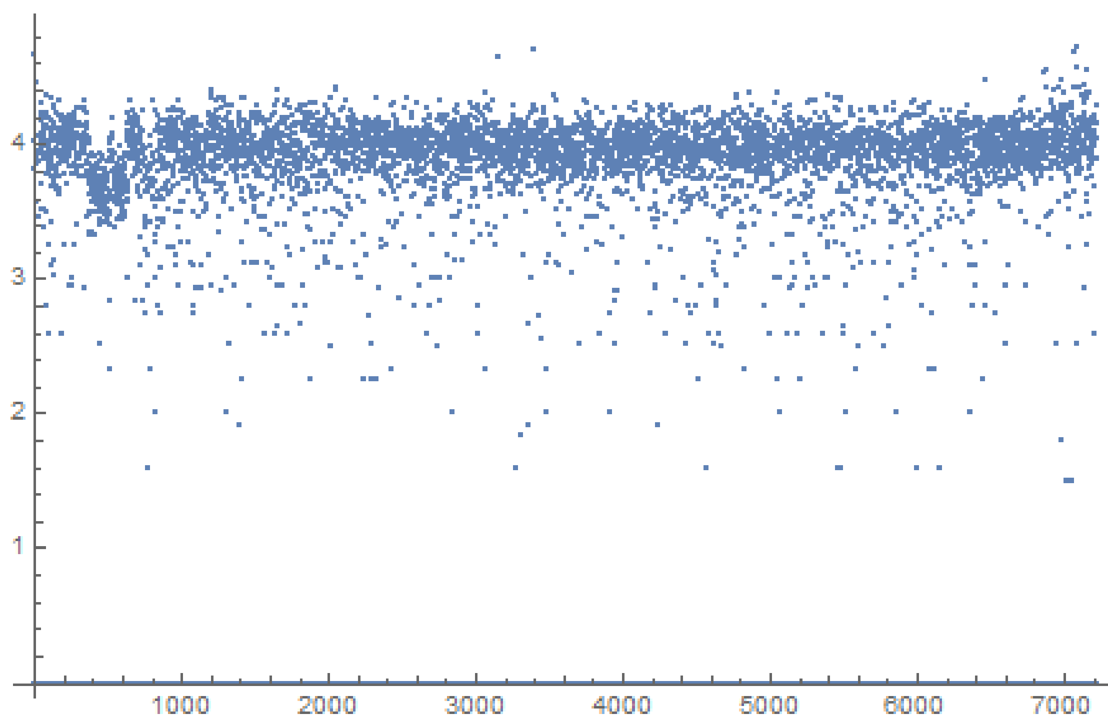
```
In[]:= ListPlot[Entropies]
```



La plupart des valeurs sont entre 5.0 et 5.4. Ceci est un signe que les données sont compressées et/ou chiffrées

Pour comprendre la variance, calculons l'entropie de toutes les liens du livre de Conan Doyle *The Hound of the Baskervilles* :

```
In[]:= BaskervillesLines = Import["http://www.gutenberg.org/cache/epub⤦
    ↳ /2852/pg2852.txt", "List"];

In[]:= EntropiesT = Map[N[Entropy[2, #]] &, BaskervillesLines];

In[]:= Variance[EntropiesT]
Out[]= 2.73883

In[]:= ListPlot[EntropiesT]
```



La plupart des valeurs sont regroupées autour de 4, mais il y a aussi des valeurs qui sont plus petites, et elles influencent la valeur finale de la variance.

Peut-être que les chaînes courtes ont une entropie plus petite, prenons les chaînes courtes du livre de Conan Doyle.

```
In[]:= Entropy[2, "Yes, sir."] // N
Out[]= 2.9477
```

Essayons encore plus petit:

```
In[]:= Entropy[2, "Yes"] // N
Out[]= 1.58496

In[]:= Entropy[2, "No"] // N
Out[]= 1.
```

8.9.2 Est-ce que les données sont compressées?

OK, donc nos données sont compressées et/ou chiffrées. Sont-elles compressées? Presque tous les compresseurs de données ajoutent un entête au début, une signature ou quelque chose comme ça. Comme on peut le voir, il n'y a pas de motifs communs au début de chaque bloc. Il est toujours possible qu'il s'agisse d'un compresseur de données écrit à la main, mais c'est très rare. D'un autre côté, les algorithmes de chiffrement maison sont plus répandus, car il est facile d'en faire un. Même des systèmes de chiffrement sans clef primitifs comme *memfrob()*[20] et ROT13 fonctionnent bien sans erreur. C'est un gros défi d'écrire un compresseur depuis zéro, en utilisant seulement sa fantaisie et son imagination de façon à ce qu'il n'ait pas de bugs évidents. Certains programmeurs implémentent des fonctions de compression de données en lisant des livres, mais ceci est aussi rare. Les deux moyens les plus fréquents sont: 1) utiliser simplement la bibliothèque open-source zlib; 2) copier/coller quelque chose de quelque part. Les algorithmes de compression open-source mettent en général une sorte d'en-tête, ainsi que les algorithmes de sites comme http://www.codeproject.com/.

8.9.3 Est-ce que les données sont chiffrées?

Les algorithmes majeurs de chiffrement de données traitent les données par bloc. DES—8 octets, AES—16 octets. Si le buffer en entrée n'est pas divisible par la taille du bloc, des zéros sont ajoutés (ou quelque chose d'autre), afin que les donnés chiffrées soient alignées sur la taille du bloc de l'algorithme. Ce n'est pas notre cas.

En utilisant Wolfram Mathematica, j'ai analysé la longueur des blocs:

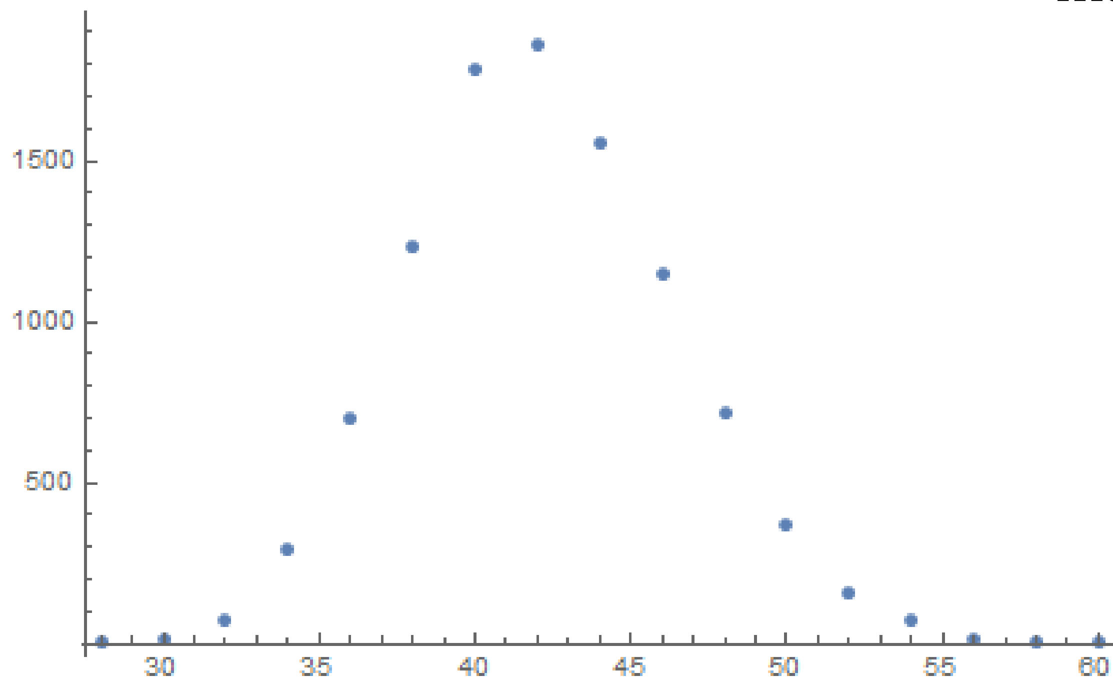
```
In[]:= Counts[Map[StringLength[#] &, BinaryStrings]]
Out[]= <|42 -> 1858, 38 -> 1235, 36 -> 699, 46 -> 1151, 40 -> 1784,
 44 -> 1558, 50 -> 366, 34 -> 291, 32 -> 74, 56 -> 15, 48 -> 716,
 30 -> 13, 52 -> 156, 54 -> 71, 60 -> 3, 58 -> 6, 28 -> 4|>
```

1858 blocs ont une taille de 42 octets, 1235 blocs ont une taille de 38 octets, etc.

J'ai fait un graphe:

```
ListPlot[Counts[Map[StringLength[#] &, BinaryStrings]]]
```

[20] http://linux.die.net/man/3/memfrob



Donc, la plupart des blocs ont une taille entre ~ 36 et ~ 48. Il y a un autre chose à remarquer: tous les blocs ont une taille paire. Pas un bloc n'a une taille impaire.

Il y a, toutefois, des flux de chiffrement qui opèrent au niveau de l'octet ou même du bit.

8.9.4 CryptoPP

Le programme qui peut parcourir cette base de données chiffrées est écrit en C# et le code .NET est fortement obscurci. Néanmoins, il y a une DLL avec du code x86, qui, après un bref examen, contient des parties de la bibliothèque open-source connue CryptoPP! (J'ai juste repéré des chaînes «CryptoPP» dedans.) Maintenant, c'est très facile de trouver toutes les fonctions à l'intérieur de la DLL car la bibliothèque CryptoPP est open-source.

La bibliothèque CryptoPP contient beaucoup de fonctions de chiffrement, AES inclus (AKA Rijndael). Les CPUs x86 récents possèdent des instructions dédiées à AES comme AESENC, AESDEC et AESKEYGENASSIST[21]. Elles ne font pas le chiffrement/déchiffrement complètement, mais elles font une part significative du travail. Et les nouvelles versions de CryptoPP les utilisent. Par exemple, ici: 1, 2. À ma surprise, lors du déchiffrement, AESENC est exécutée, tandis que AESDEC ne l'est pas (j'ai vérifié avec mon utilitaire tracer, mais n'importe quel débogueur peut être utilisé). J'ai vérifié, si mon CPU supporte réellement les instructions AES. Certains CPUs Intel i3 ne les supportent pas. Et si non, la bibliothèque CryptoPP se rabat sur les fonctions

[21]https://en.wikipedia.org/wiki/AES_instruction_set

implémentées de l'ancienne façon [22]. Mais mon CPU les supporte. Pourquoi AESDEC n'est pas exécuté? Pourquoi le programme utilise le chiffrement AES pour déchiffrer la base de données?

OK, ce n'est pas un problème de trouver la fonction qui chiffre les blocs. Elle est appelée
CryptoPP::Rijndael::Enc::ProcessAndXorBlock : src, et elle peut être appelée depuis une autre fonction:
Rijndael::Enc::AdvancedProcessBlocks() src, qui, à son tour, appelle les deux fonctions: (AESNI_Enc_Block et AESNI_Enc_4_Blocks) qui ont les instructions AESENC.

Donc, a en juger par les entrailles de CryptoPP
CryptoPP::Rijndael::Enc::ProcessAndXorBlock() chiffre un bloc 16-octet. Mettons un point d'arrêt dessus et voyons ce qui se produit pendant le déchiffrement. J'utilise à nouveau mon petit outil tracer. Le logiciel doit déchiffrer le premier bloc de données maintenant. Oh, à propos, voici le premier bloc de données converti de l'encodage en base64 vers des données hexadécimale, faisons le manuellement:

```
00000000: CA 39 B1 85 75 1B 84 1F  F9 31 5E 39 72 13 EC 5D  .9..u....1^9r↙
    ↳ ..]
00000010: 95 80 27 02 21 D5 2D 1A  0F D9 45 9F 75 EE 24 C4  ..'.!.-...E.u.$↙
    ↳ .
00000020: B1 27 7F 84 FE 41 37 86  C9 C0                    .'...A7...
```

Voici les arguments de la fonction d'après les fichiers sources de CryptoPP:

```
size_t Rijndael::Enc::AdvancedProcessBlocks(const byte *inBlocks, const ↙
    ↳ byte *xorBlocks, byte *outBlocks, size_t length, word32 flags);
```

Donc, il y a 5 arguments. Les flags possibles sont:

```
enum {BT_InBlockIsCounter=1, BT_DontIncrementInOutPointers=2, BT_XorInput↙
    ↳ =4, BT_ReverseDirection=8, BT_AllowParallel=16} ↙
    ↳ FlagsForAdvancedProcessBlocks;
```

OK, lançons tracer sur la fonction *ProcessAndXorBlock()* :

```
... tracer.exe -l:filename.exe bpf=filename.exe!0x4339a0,args:5,dump_args:0↙
    ↳ x10

Warning: no tracer.cfg file.
PID=1984|New process software.exe
no module registered with image base 0x77320000
no module registered with image base 0x76e20000
no module registered with image base 0x77320000
no module registered with image base 0x77220000
Warning: unknown (to us) INT3 breakpoint at ntdll.dll!↙
    ↳ LdrVerifyImageMatchesChecksum+0x96c (0x776c103b)
(0) software.exe!0x4339a0(0x38b920, 0x0, 0x38b978, 0x10, 0x0) (called from ↙
    ↳ software.exe!.text+0x33c0d (0x13e4c0d))
Argument 1/5
```

[22]https://github.com/mmoss/cryptopp/blob/2772f7b57182b31a41659b48d5f35a7b6cedd34d/src/rijndael.cpp#L355

```
0038B920: 01 00 00 00 FF FF FF FF-79 C1 69 0B 67 C1 04 7D "........y.i.g⤦
    ↳ ..}"
Argument 3/5
0038B978: CD CD CD CD CD CD CD CD-CD CD CD CD CD CD CD CD ⤦
    ↳ "................"
(0) software.exe!0x4339a0() -> 0x0
Argument 3/5 difference
00000000: C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N⤦
    ↳ {3....1.99..]"
(0) software.exe!0x4339a0(0x38a828, 0x38a838, 0x38bb40, 0x0, 0x8) (called ⤦
    ↳ from software.exe!.text+0x3a407 (0x13eb407))
Argument 1/5
0038A828: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "..'.!.-...E.u.$⤦
    ↳ ."
Argument 2/5
0038A838: B1 27 7F 84 FE 41 37 86-C9 C0 00 CD CD CD CD CD ".'...A7⤦
    ↳ ........."
Argument 3/5
0038BB40: CD CD CD CD CD CD CD CD-CD CD CD CD CD CD CD CD ⤦
    ↳ "................"
(0) software.exe!0x4339a0() -> 0x0
(0) software.exe!0x4339a0(0x38b920, 0x38a828, 0x38bb30, 0x10, 0x0) (called ⤦
    ↳ from software.exe!.text+0x33c0d (0x13e4c0d))
Argument 1/5
0038B920: CA 39 B1 85 75 1B 84 1F-F9 31 5E 39 72 13 EC 5D ".9..u....1^9r⤦
    ↳ ..]"
Argument 2/5
0038A828: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "..'.!.-...E.u.$⤦
    ↳ ."
Argument 3/5
0038BB30: CD CD CD CD CD CD CD CD-CD CD CD CD CD CD CD CD ⤦
    ↳ "................"
(0) software.exe!0x4339a0() -> 0x0
Argument 3/5 difference
00000000: 45 00 20 00 4A 00 4F 00-48 00 4E 00 53 00 00 00 "E. .J.O.H.N.S⤦
    ↳ ..."
(0) software.exe!0x4339a0(0x38b920, 0x0, 0x38b978, 0x10, 0x0) (called from ⤦
    ↳ software.exe!.text+0x33c0d (0x13e4c0d))
Argument 1/5
0038B920: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "..'.!.-...E.u.$⤦
    ↳ ."
Argument 3/5
0038B978: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "..'.!.-...E.u.$⤦
    ↳ ."
(0) software.exe!0x4339a0() -> 0x0
Argument 3/5 difference
00000000: B1 27 7F E4 9F 01 E3 81-CF C6 12 FB B9 7C F1 BC ⤦
    ↳ ".'..........|.."
PID=1984|Process software.exe exited. ExitCode=0 (0x0)
```

Ici nous pouvons voir l'entrée de la fonction *ProcessAndXorBlock()*, et sa sortie.

Ceci est la sortie de la fonction lors du premier appel:

```
00000000: C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N↙
    ↳ {3....1.99..]"
```

Puis la fonction *ProcessAndXorBlock()* est appelée avec un bloc de longueur zéro, mais avec le flag 8 (*BT_ReverseDirection*).

Second appel:

```
00000000: 45 00 20 00 4A 00 4F 00-48 00 4E 00 53 00 00 00 "E. .J.O.H.N.S↙
    ↳ ..."
```

Maintenant, il y a des chaînes qui nous sont familières!

Troisième appel:

```
00000000: B1 27 7F E4 9F 01 E3 81-CF C6 12 FB B9 7C F1 BC ↙
    ↳ ".'..........|.."
```

La première sortie est très similaire aux 16 premiers octets du buffer chiffré.

Sortie du premier appel à *ProcessAndXorBlock()* :

```
00000000: C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N↙
    ↳ {3....1.99..]"
```

16 premiers octets du buffer chiffré:

```
00000000: CA 39 B1 85 75 1B 84 1F F9 31 5E 39 72 13 EC 5D  .9..u....1^9r..]
```

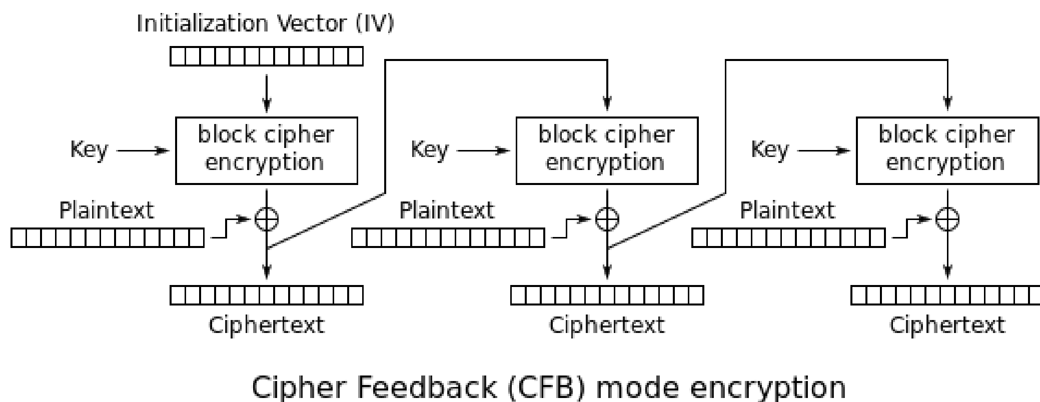
Il y a trop d'octets égaux! Comment le résultat du chiffrement AES peut-il être aussi similaire au buffer chiffré alors que ceci n'est pas du chiffrement mais bien du déchiffrement?!

8.9.5 Mode Cipher Feedback

La réponse est CFB[23] : Dans ce mode, l'algorithme AES n'est pas utilisé comme un algorithme de chiffrement, mais comme un dispositif qui génère des données aléatoires cryptographiquement sûres. Le chiffrement effectif est obtenu en utilisant une simple opération XOR.

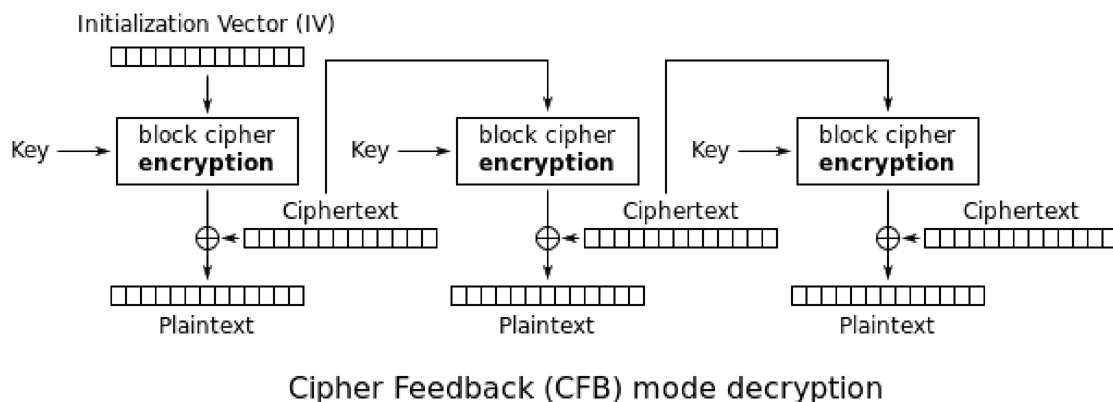
Voici l'algorithme de chiffrement (les images proviennent de Wikipédia) :

[23]Cipher Feedback



Cipher Feedback (CFB) mode encryption

Et le déchiffrement:



Cipher Feedback (CFB) mode decryption

Maintenant regardons: le chiffrement AES génère 16 octets (ou 128 bits) de données *aléatoires* destinées à être utilisées lors du XOR, qui nous oblige à utiliser tous les 16 octets? Si à la dernière itération nous n'avons qu'un octet de données, nous ne chiffrons qu'un octet avec un octet de données *aléatoires* générée. Ceci conduit à une propriété importante du mode CFB : les données ne doivent pas être adaptées à une taille, des données de taille arbitraire peuvent être chiffrées et déchiffrées.

Oh, c'est pour ça que les blocs chiffrés ne sont pas complétés. Et c'est pourquoi l'instruction AESDEC n'est jamais appelée.

Essayons de déchiffrer le premier bloc manuellement, en utilisant Python. Le mode CFB utilise aussi un IV, comme *semence* pour CSPRNG[24]. Dans notre cas, l'IV est le bloc qui est chiffré à la première itération:

[24]Cryptographically Secure Pseudorandom Number Generator (générateur de nombres pseudo-aléatoire cryptographiquement sûr)

```
0038B920: 01 00 00 00 FF FF FF FF-79 C1 69 0B 67 C1 04 7D "........y.i.g↙
    ↳ ..}"
```

Oh, et nous devons aussi retrouver la clef de chiffrement. Il y a AESKEYGENASSIST dans la DLL, et elle est appelée, et elle est utilisée dans la fonction src. C'est facile de la trouver dans IDA et de mettre un point d'arrêt. Voyons:

```
... tracer.exe -l:filename.exe bpf=filename.exe!0x435c30,args:3,dump_args:0↙
    ↳ x10

Warning: no tracer.cfg file.
PID=2068|New process software.exe
no module registered with image base 0x77320000
no module registered with image base 0x76e20000
no module registered with image base 0x77320000
no module registered with image base 0x77220000
Warning: unknown (to us) INT3 breakpoint at ntdll.dll!↙
    ↳ LdrVerifyImageMatchesChecksum+0x96c (0x776c103b)
(0) software.exe!0x435c30(0x15e8000, 0x10, 0x14f808) (called from software.↙
    ↳ exe!.text+0x22fa1 (0x13d3fa1))
Argument 1/3
015E8000: CD C5 7E AD 28 5F 6D E1-CE 8F CC 29 B1 21 88 8E "..~.(_m....)↙
    ↳ .!.."
Argument 3/3
0014F808: 38 82 58 01 C8 B9 46 00-01 D1 3C 01 00 F8 14 00 "8.X...F↙
    ↳ ...<....."
Argument 3/3 +0x0: software.exe!.rdata+0x5238
Argument 3/3 +0x8: software.exe!.text+0x1c101
(0) software.exe!0x435c30() -> 0x13c2801
PID=2068|Process software.exe exited. ExitCode=0 (0x0)
```

Donc, ceci est la clef: *CD C5 7E AD 28 5F 6D E1-CE 8F CC 29 B1 21 88 8E*.

Durant le déchiffrement manuel, nous obtenons ceci:

```
00000000: 0D 00 FF FE 46 00 52 00  41 00 4E 00 4B 00 49 00  ....F.R.A.N.K.I↙
    ↳ .
00000010: 45 00 20 00 4A 00 4F 00  48 00 4E 00 53 00 66 66  E. .J.O.H.N.S.↙
    ↳ ff
00000020: 66 66 66 9E 61 40 D4 07  06 01                    fff.a@....
```

Maintenant, c'est quelque chose de lisible! Et nous comprenons pourquoi il y avait autant d'octets égaux dans la première itération de déchiffrement: car le text en clair a beaucoup d'octet à zéro! Déchiffrons le second bloc:

```
00000000: 17 98 D0 84 3A E9 72 4F  DB 82 3F AD E9 3E 2A A8  ....:.rO↙
    ↳ ..?..>*.
00000010: 41 00 52 00 52 00 4F 00  4E 00 CD CC CC CC CC CC  A.R.R.O.N↙
    ↳ .......
00000020: 1B 40 D4 07 06 01                                 .@....
```

Les troisième, quatrième et cinquième:

```
00000000: 5D 90 59 06 EF F4 96 B4  7C 33 A7 4A BE FF 66 AB  ].Y.....|3.J..f↙
    ↳ .
00000010: 49 00 47 00 47 00 53 00  00 00 00 00 00 C0 65 40  I.G.G.S.......↙
    ↳ e@
00000020: D4 07 06 01                                       ....
```

```
00000000: D3 15 34 5D 21 18 7C 6E  AA F8 2D FE 38 F9 D7 4E  ..4]!.|n..-.8..↙
    ↳ N
00000010: 41 00 20 00 44 00 4F 00  48 00 45 00 52 00 54 00  A. .D.O.H.E.R.T↙
    ↳ .
00000020: 59 00 48 E1 7A 14 AE FF  68 40 D4 07 06 02        Y.H.z...h@....
```

```
00000000: 1E 8B 90 0A 17 7B C5 52  31 6C 4E 2F DE 1B 27 19  .....{.R1lN↙
    ↳ ...'.
00000010: 41 00 52 00 43 00 55 00  53 00 00 00 00 00 00 60  A.R.C.U.S↙
    ↳ .......
00000020: 66 40 D4 07 06 03                                 f@....
```

Tous les blocs déchiffrés semblent correct, à l'exception des 16 premiers octets.

8.9.6 Initializing Vector

Qu'est-ce qui peut affecter les 16 premiers octets?

Revenons à nouveau à l'algorithme de déchiffrement CFB : 8.9.5 on page 1120.

Nous pouvons voir que l'IV peut affecter le déchiffrement de la première opération de déchiffrement, mais pas la seconde, car lors de la seconde itération, le texte chiffré de la première itération est utilisé, et en cas de déchiffrement, c'est le même, quelque soit l'IV !

Donc, l'IV est sans doute différent à chaque fois. En utilisant mon tracer, j'ai regardé la première entrée lors du déchiffrement du second bloc du fichier XML :

```
0038B920: 02 00 00 00 FE FF FF FF-79 C1 69 0B 67 C1 04 7D "........y.i.g↙
    ↳ ..}"
```

...troisième:

```
0038B920: 03 00 00 00 FD FF FF FF-79 C1 69 0B 67 C1 04 7D "........y.i.g↙
    ↳ ..}"
```

Il semble que le premier et le cinquième octet changent à chaque fois. J'en ai finalement conclu que le premier entier 32-bit est simplement OrderID du fichier XML, et le second entier 32-bit est aussi OrderID, mais multiplié par -1. Tous les 8 autres octets sont les mêmes pour chaque opération. Maintenant, j'ai déchiffré la base de données entière: `https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted_DB1/decrypted.full.txt`.

Le script Python utilisé pour ceci est: `https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted_DB1/decrypt_blocks.py`.

Peut-être que l'auteur voulait chiffrer chaque bloc différemment, donc il a utilisé OrderID comme une partie de la clef. Il aurait aussi été possible de créer une clef AES différente, au lieu de l'IV.

Dinc maintenant nous savons que l'IV affecte seulement le premier bloc lors du déchiffrement en mode CFB, ceci en est une caractéristique. Tous les autres blocs peuvent être déchiffrés sans connaître l'IV, mais en utilisant la clef.

OK, donc pourquoi le mode CFB ? Apparemment, parce que le tout premier exemple sur le wiki de CryptoPP utilise le mode CFB : http://www.cryptopp.com/wiki/Advanced_Encryption_Standard#Encrypting_and_Decrypting_Using_AES. On peut aussi supposer que le développeur l'a choisi pour sa simplicité: l'exemple peut chiffrer/déchiffrer des chaînes de texte de longueur arbitraire, sans remplissage.

Il est aussi probable que l'auteur du programme a juste copié/collé l'exemple depuis la page wiki de CryptoPP. Beaucoup de programmeurs font ça.

La seule différence est que l'IV est choisi aléatoirement dans l'exemple du wiki de CryptoPP, alors que cet indéterminisme n'était pas permis aux programmeurs du logiciel que nous disséquons maintenant, donc ils ont choisi d'initialiser l'IV en utilisant OrderID.

Nous pouvons maintenant procéder à l'analyse du cas de chaque octet dans le bloc déchiffré.

8.9.7 Structure du buffer

Prenons les quatre premier bloc déchiffrés:

```
00000000: 0D 00 FF FE 46 00 52 00  41 00 4E 00 4B 00 49 00  ....F.R.A.N.K.I↙
    ↳ .
00000010: 45 00 20 00 4A 00 4F 00  48 00 4E 00 53 00 66 66  E. .J.O.H.N.S.↙
    ↳ ff
00000020: 66 66 66 9E 61 40 D4 07  06 01                    fff.a@....

00000000: 0B 00 FF FE 4C 00 4F 00  52 00 49 00 20 00 42 00  ....L.O.R.I. .B↙
    ↳ .
00000010: 41 00 52 00 52 00 4F 00  4E 00 CD CC CC CC CC CC  A.R.R.O.N↙
    ↳ .......
00000020: 1B 40 D4 07 06 01                                 .@....

00000000: 0A 00 FF FE 47 00 41 00  52 00 59 00 20 00 42 00  ....G.A.R.Y. .B↙
    ↳ .
00000010: 49 00 47 00 47 00 53 00  00 00 00 00 00 C0 65 40  I.G.G.S.......↙
    ↳ e@
00000020: D4 07 06 01                                       ....

00000000: 0F 00 FF FE 4D 00 45 00  4C 00 49 00 4E 00 44 00  ....M.E.L.I.N.D↙
    ↳ .
00000010: 41 00 20 00 44 00 4F 00  48 00 45 00 52 00 54 00  A. .D.O.H.E.R.T↙
    ↳ .
00000020: 59 00 48 E1 7A 14 AE FF  68 40 D4 07 06 02        Y.H.z...h@....
```

On voit clairement des chaînes de textes encodées en UTF-16, ce sont les noms et noms de famille. Le premier octet (ou mot de 16-bit) semble être la longueur de la chaîne, nous pouvons vérifier visuellement. *FF FE* semble être le BOM Unicode.

Il y a 12 autres octets après chaque chaîne.

En utilisant ce script (https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted_DB1/dump_buffer_rest.py) j'ai obtenu une sélection aléatoire de *fins* (de bloc) :

```
dennis@...:$ python decrypt.py encrypted.xml | shuf | head -20
00000000: 48 E1 7A 14 AE 5F 62 40  DD 07 05 08              H.z.._b@....
00000000: 00 00 00 00 00 40 5A 40  DC 07 08 18              .....@Z@....
00000000: 00 00 00 00 00 80 56 40  D7 07 0B 04              ......V@....
00000000: 00 00 00 00 00 60 61 40  D7 07 0C 1C              ......a@....
00000000: 00 00 00 00 00 20 63 40  D9 07 05 18              ..... c@....
00000000: 3D 0A D7 A3 70 FD 34 40  D7 07 07 11              =...p.4@....
00000000: 00 00 00 00 00 A0 63 40  D5 07 05 19              ......c@....
00000000: CD CC CC CC CC 3C 5C 40  D7 07 08 11              .......@....
00000000: 66 66 66 66 66 FE 62 40  D4 07 06 05              fffff.b@....
00000000: 1F 85 EB 51 B8 FE 40 40  D6 07 09 1E              ...Q..@@....
00000000: 00 00 00 00 00 40 5F 40  DC 07 02 18              .....@_@....
00000000: 48 E1 7A 14 AE 9F 67 40  D8 07 05 12              H.z...g@....
00000000: CD CC CC CC CC 3C 5E 40  DC 07 01 07              ......^@....
00000000: 00 00 00 00 00 00 67 40  D4 07 0B 0E              ......g@....
00000000: 00 00 00 00 00 40 51 40  DC 07 04 0B              .....@Q@....
00000000: 00 00 00 00 00 40 56 40  D7 07 07 0A              .....@V@....
00000000: 8F C2 F5 28 5C 7F 55 40  DB 07 01 16              ...(..U@....
00000000: 00 00 00 00 00 00 32 40  DB 07 06 09              ......2@....
00000000: 66 66 66 66 66 7E 66 40  D9 07 0A 06              fffff~f@....
00000000: 48 E1 7A 14 AE DF 68 40  D5 07 07 16              H.z...h@....
```

Nous voyons tout d'abord que les octets 0x40 et 0x07 sont présent dans chaque *fin*. Le tout dernier octet est toujours dans l'intervalle 1..0x1F (1..31), j'ai vérifié. Le pénultième octet est toujours dans l'intervalle 1..0xC (1..12). Ouah, ça ressemble à une date! L'année peut être représentée comme une valeur 16-bit, et peut-être que les 4 derniers octets sont une date (16 bits pour l'année, 8 bits pour le mois et les 8 restants pour le jour)? 0x7DD est 2013, 0x7D5 est 2005, etc. Ça semble juste. Ceci est une date. Il y a 8 octets supplémentaires. À en juger par le fait que ceci est une base de données appelée *orders*, peut-être s'agit-il d'une sorte de somme ici? J'ai essayé de les interpréter comme des réels en double précision IEEE 754 et ai affiché toutes les valeurs!

Certaines sont:

```
71.0
134.0
51.95
53.0
121.99
96.95
98.95
15.95
85.95
```

```
184.99
94.95
29.95
85.0
36.0
130.99
115.95
87.99
127.95
114.0
150.95
```

Ça ressemble à des nombres réels

Maintenant, nous pouvons afficher les noms, sommes et dates.

```
plain:
00000000: 0D 00 FF FE 46 00 52 00  41 00 4E 00 4B 00 49 00  ....F.R.A.N.K.I⤦
    ↳ .
00000010: 45 00 20 00 4A 00 4F 00  48 00 4E 00 53 00 66 66  E. .J.O.H.N.S.⤦
    ↳ ff
00000020: 66 66 66 9E 61 40 D4 07  06 01                    fff.a@....
OrderID= 1 name= FRANKIE JOHNS sum= 140.95 date= 2004 / 6 / 1

plain:
00000000: 0B 00 FF FE 4C 00 4F 00  52 00 49 00 20 00 42 00  ....L.O.R.I. .B⤦
    ↳ .
00000010: 41 00 52 00 52 00 4F 00  4E 00 CD CC CC CC CC CC  A.R.R.O.N⤦
    ↳ .......
00000020: 1B 40 D4 07 06 01                                 .@....
OrderID= 2 name= LORI BARRON sum= 6.95 date= 2004 / 6 / 1

plain:
00000000: 0A 00 FF FE 47 00 41 00  52 00 59 00 20 00 42 00  ....G.A.R.Y. .B⤦
    ↳ .
00000010: 49 00 47 00 47 00 53 00  00 00 00 00 00 C0 65 40  I.G.G.S.......⤦
    ↳ e@
00000020: D4 07 06 01                                       ....
OrderID= 3 name= GARY BIGGS sum= 174.0 date= 2004 / 6 / 1

plain:
00000000: 0F 00 FF FE 4D 00 45 00  4C 00 49 00 4E 00 44 00  ....M.E.L.I.N.D⤦
    ↳ .
00000010: 41 00 20 00 44 00 4F 00  48 00 45 00 52 00 54 00  A. .D.O.H.E.R.T⤦
    ↳ .
00000020: 59 00 48 E1 7A 14 AE FF  68 40 D4 07 06 02        Y.H.z...h@....
OrderID= 4 name= MELINDA DOHERTY sum= 199.99 date= 2004 / 6 / 2

plain:
00000000: 0B 00 FF FE 4C 00 45 00  4E 00 41 00 20 00 4D 00  ....L.E.N.A. .M⤦
    ↳ .
00000010: 41 00 52 00 43 00 55 00  53 00 00 00 00 00 00 60  A.R.C.U.S⤦
    ↳ .......
00000020: 66 40 D4 07 06 03                                 f@....
```

```
OrderID= 5 name= LENA MARCUS sum= 179.0 date= 2004 / 6 / 3
```

En voir plus: https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted_DB1/decrypted.full.with_data.txt. Ou filtré: https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted_DB1/decrypted.short.txt. Ça semble correct.

Ceci est une sorte de sérialisation POO, i.e., stockant différents types de valeurs dans un buffer binaire pour le stocker et/ou le transmettre.

8.9.8 Bruit en fin de buffer

La seule question qui reste est que, parfois, la *fin* est plus longue:

```
00000000: 0E 00 FF FE 54 00 48 00  45 00 52 00 45 00 53 00  ....T.H.E.R.E.S↙
    ↳ .
00000010: 45 00 20 00 54 00 55 00  54 00 54 00 4C 00 45 00  E. .T.U.T.T.L.E↙
    ↳ .
00000020: 66 66 66 66 66 1E 63 40  D4 07 07 1A 00 07 07 19  fffff.c@↙
    ↳ ........
OrderID= 172 name= THERESE TUTTLE sum= 152.95 date= 2004 / 7 / 26
```

(Les octets *00 07 07 19* ne sont pas utilisés et servent de remplissage.)

```
00000000: 0C 00 FF FE 4D 00 45 00  4C 00 41 00 4E 00 49 00  ....M.E.L.A.N.I↙
    ↳ .
00000010: 45 00 20 00 4B 00 49 00  52 00 4B 00 00 00 00 00  E. .K.I.R.K↙
    ↳ .....
00000020: 00 20 64 40 D4 07 09 02  00 02                    . d@......
OrderID= 286 name= MELANIE KIRK sum= 161.0 date= 2004 / 9 / 2
```

(*00 02* ne sont pas utilisés.)

Après un examen rigoureux, on peut voir que le but à la fin de la *fin* est juste le reste d'un chiffrement précédent!

Voici deux buffers consécutifs:

```
00000000: 10 00 FF FE 42 00 4F 00  4E 00 4E 00 49 00 45 00  ....B.O.N.N.I.E↙
    ↳ .
00000010: 20 00 47 00 4F 00 4C 00  44 00 53 00 54 00 45 00   .G.O.L.D.S.T.E↙
    ↳ .
00000020: 49 00 4E 00 9A 99 99 99  99 79 46 40 D4 07 07 19  I.N......yF@↙
    ↳ ....
OrderID= 171 name= BONNIE GOLDSTEIN sum= 44.95 date= 2004 / 7 / 25

00000000: 0E 00 FF FE 54 00 48 00  45 00 52 00 45 00 53 00  ....T.H.E.R.E.S↙
    ↳ .
00000010: 45 00 20 00 54 00 55 00  54 00 54 00 4C 00 45 00  E. .T.U.T.T.L.E↙
    ↳ .
00000020: 66 66 66 66 66 1E 63 40  D4 07 07 1A 00 07 07 19  fffff.c@↙
    ↳ ........
OrderID= 172 name= THERESE TUTTLE sum= 152.95 date= 2004 / 7 / 26
```

(Les derniers octets *07 07 19* sont copiés du buffer précédent.)

Un autre exemple de deux buffers consécutifs:

```
00000000: 0D 00 FF FE 4C 00 4F 00  52 00 45 00 4E 00 45 00  ....L.O.R.E.N.E↙
    ↳ .
00000010: 20 00 4F 00 54 00 4F 00  4F 00 4C 00 45 00 CD CC   .O.T.O.O.L.E↙
    ↳ ...
00000020: CC CC CC 3C 5E 40 D4 07  09 02                    ...<^@....
OrderID= 285 name= LORENE OTOOLE sum= 120.95 date= 2004 / 9 / 2

00000000: 0C 00 FF FE 4D 00 45 00  4C 00 41 00 4E 00 49 00  ....M.E.L.A.N.I↙
    ↳ .
00000010: 45 00 20 00 4B 00 49 00  52 00 4B 00 00 00 00 00  E. .K.I.R.K↙
    ↳ .....
00000020: 00 20 64 40 D4 07 09 02  00 02                    . d@......
OrderID= 286 name= MELANIE KIRK sum= 161.0 date= 2004 / 9 / 2
```

Le dernier octet 02 a été copié du buffer en texte clair précédent.

C'est possible si le buffer utilisé lors du chiffrement est global et/ou s'il n'est pas mis à zéro entre chaque chiffrement. La taille du buffer final est aussi chaotique, néanmoins, le bogue reste sans conséquence car il n'affecte pas le processus de déchiffrement, qui ignore le bruit à la fin. C'est une erreur courante. Il était présent dans OpenSSL (Heartbleed bug).

8.9.9 Conclusion

Résumé: Chaque rétro-ingénieur pratiquant doit être familier avec la majorité des algorithmes ainsi que la majorité des modes de chiffrement. Quelques livres à ce sujet: 12.1.10 on page 1316.

Le contenu *chiffré* de la base de données a été artificiellement construit par moi, pour les besoins de la démonstration. J'ai obtenu les nom et noms de famille les plus répandus au USA ici: http://stackoverflow.com/questions/1803628/raw-list-of-person-names, et les ai combiné aléatoirement. Les dates et montants ont aussi été générés aléatoirement.

Tous les fichiers utilisés dans cette partie sont ici: https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted_DB1.

Néanmoins, j'ai observé de telles caractéristiques dans des logiciels réels. Cet exemple est basé dessus.

8.9.10 Post Scriptum: brute-force IV

Le cas que vous venez de voir a été construit artificiellement, mais il est basé sur des logiciels réels que j'ai rétro-ingénièré. Lorsque j'ai travaillé dessus, j'ai d'abord remarqué que l'IV avait été généré en utilisant un nombre 32-bit, et je n'ai pas été capable de trouver un lien entre cette valeur et OrderID. Donc j'ai utilisé le brute-force, ce qui est aussi possible ici.

Ce n'est pas un problème d'énumérer toutes les valeurs 32-bit et d'essayer chacune d'elle comme base pour l'IV. Ensuite vous déchiffrez le premier bloc de 16 octets et vérifiez les octets à zéro, qui sont toujours à des places fixes.

8.10 Overclocker le mineur de Bitcoin Cointerra

Il y avait le mineur de Bitcoin Cointerra, ressemblant à ceci:

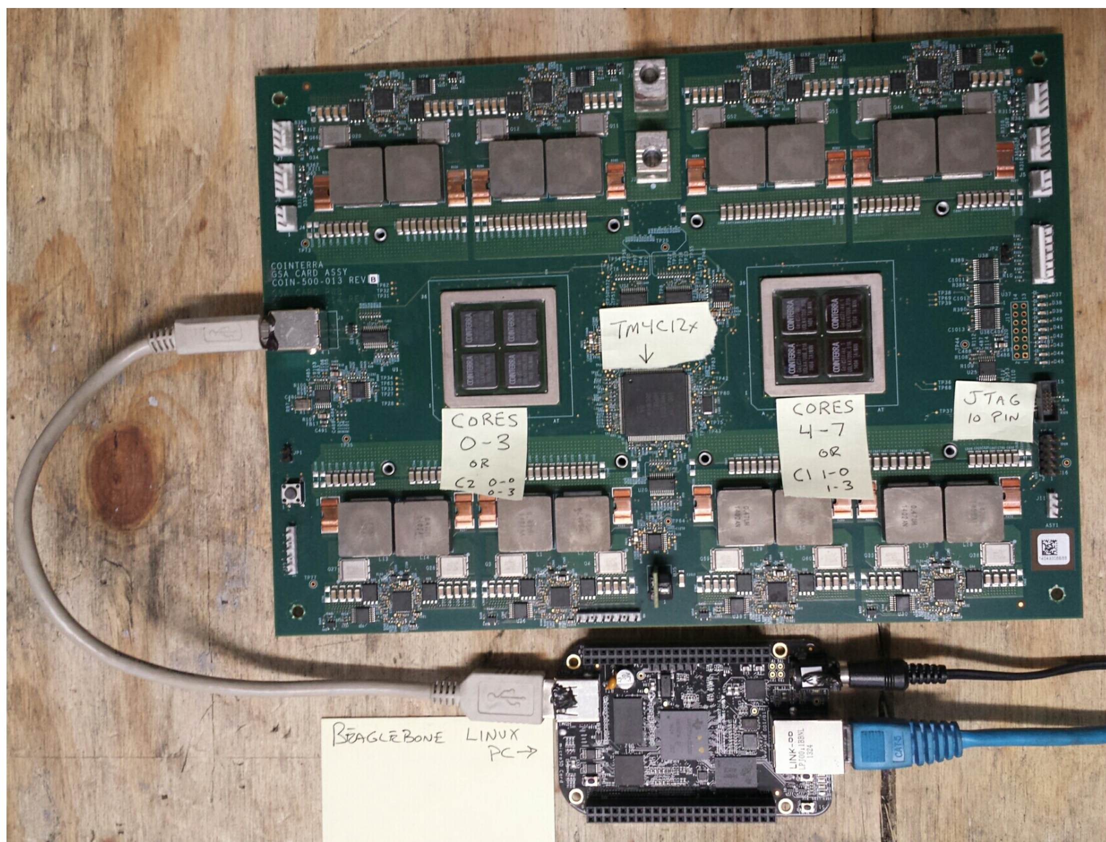


Fig. 8.14: Carte

Et il y avait aussi (peut-être leaké) l'utilitaire[25] qui peut définir la fréquence d'horloge pour la carte. Il fonctionne sur une carte additionnelle BeagleBone Linux ARM (petite carte en bas de l'image).

Et on m'avait demandé une fois s'il est possible de modifier cet utilitaire pour voir quelles sont les fréquences qui peuvent être définies, et celles qui ne peuvent pas l'être. Et est-il possible de l'ajuster?

[25]Peut être téléchargé ici: https://beginners.re/paywall/RE4B-source/current-tree//examples/bitcoin_miner/files/cointool-overclock

L'utilitaire doit être exécuté comme cela: `./cointool-overclock 0 0 900`, où 900 est la fréquence en MHz. Si la fréquence est trop grande, l'utilitaire affiche «Error with arguments» et se termine.

Ceci est le morceau de code autour de la référence à la chaîne de texte «Error with arguments» :

```
...

.text:0000ABC4         STR      R3, [R11,#var_28]
.text:0000ABC8         MOV      R3, #optind
.text:0000ABD0         LDR      R3, [R3]
.text:0000ABD4         ADD      R3, R3, #1
.text:0000ABD8         MOV      R3, R3,LSL#2
.text:0000ABDC         LDR      R2, [R11,#argv]
.text:0000ABE0         ADD      R3, R2, R3
.text:0000ABE4         LDR      R3, [R3]
.text:0000ABE8         MOV      R0, R3  ; nptr
.text:0000ABEC         MOV      R1, #0  ; endptr
.text:0000ABF0         MOV      R2, #0  ; base
.text:0000ABF4         BL       strtoll
.text:0000ABF8         MOV      R2, R0
.text:0000ABFC         MOV      R3, R1
.text:0000AC00         MOV      R3, R2
.text:0000AC04         STR      R3, [R11,#var_2C]
.text:0000AC08         MOV      R3, #optind
.text:0000AC10         LDR      R3, [R3]
.text:0000AC14         ADD      R3, R3, #2
.text:0000AC18         MOV      R3, R3,LSL#2
.text:0000AC1C         LDR      R2, [R11,#argv]
.text:0000AC20         ADD      R3, R2, R3
.text:0000AC24         LDR      R3, [R3]
.text:0000AC28         MOV      R0, R3  ; nptr
.text:0000AC2C         MOV      R1, #0  ; endptr
.text:0000AC30         MOV      R2, #0  ; base
.text:0000AC34         BL       strtoll
.text:0000AC38         MOV      R2, R0
.text:0000AC3C         MOV      R3, R1
.text:0000AC40         MOV      R3, R2
.text:0000AC44         STR      R3, [R11,#third_argument]
.text:0000AC48         LDR      R3, [R11,#var_28]
.text:0000AC4C         CMP      R3, #0
.text:0000AC50         BLT      errors_with_arguments
.text:0000AC54         LDR      R3, [R11,#var_28]
.text:0000AC58         CMP      R3, #1
.text:0000AC5C         BGT      errors_with_arguments
.text:0000AC60         LDR      R3, [R11,#var_2C]
.text:0000AC64         CMP      R3, #0
.text:0000AC68         BLT      errors_with_arguments
.text:0000AC6C         LDR      R3, [R11,#var_2C]
.text:0000AC70         CMP      R3, #3
.text:0000AC74         BGT      errors_with_arguments
.text:0000AC78         LDR      R3, [R11,#third_argument]
```

```
.text:0000AC7C         CMP      R3, #0x31
.text:0000AC80         BLE      errors_with_arguments
.text:0000AC84         LDR      R2, [R11,#third_argument]
.text:0000AC88         MOV      R3, #950
.text:0000AC8C         CMP      R2, R3
.text:0000AC90         BGT      errors_with_arguments
.text:0000AC94         LDR      R2, [R11,#third_argument]
.text:0000AC98         MOV      R3, #0x51EB851F
.text:0000ACA0         SMULL    R1, R3, R3, R2
.text:0000ACA4         MOV      R1, R3,ASR#4
.text:0000ACA8         MOV      R3, R2,ASR#31
.text:0000ACAC         RSB      R3, R3, R1
.text:0000ACB0         MOV      R1, #50
.text:0000ACB4         MUL      R3, R1, R3
.text:0000ACB8         RSB      R3, R3, R2
.text:0000ACBC         CMP      R3, #0
.text:0000ACC0         BEQ      loc_ACEC
.text:0000ACC4
.text:0000ACC4 errors_with_arguments
.text:0000ACC4
.text:0000ACC4         LDR      R3, [R11,#argv]
.text:0000ACC8         LDR      R3, [R3]
.text:0000ACCC         MOV      R0, R3  ; path
.text:0000ACD0         BL       __xpg_basename
.text:0000ACD4         MOV      R3, R0
.text:0000ACD8         MOV      R0, #aSErrorWithArgu ; format
.text:0000ACE0         MOV      R1, R3
.text:0000ACE4         BL       printf
.text:0000ACE8         B        loc_ADD4
.text:0000ACEC ; ---------------------------------------------------------
.text:0000ACEC
.text:0000ACEC loc_ACEC                 ; CODE XREF: main+66C
.text:0000ACEC         LDR      R2, [R11,#third_argument]
.text:0000ACF0         MOV      R3, #499
.text:0000ACF4         CMP      R2, R3
.text:0000ACF8         BGT      loc_AD08
.text:0000ACFC         MOV      R3, #0x64
.text:0000AD00         STR      R3, [R11,#unk_constant]
.text:0000AD04         B        jump_to_write_power
.text:0000AD08 ; ---------------------------------------------------------
.text:0000AD08
.text:0000AD08 loc_AD08                 ; CODE XREF: main+6A4
.text:0000AD08         LDR      R2, [R11,#third_argument]
.text:0000AD0C         MOV      R3, #799
.text:0000AD10         CMP      R2, R3
.text:0000AD14         BGT      loc_AD24
.text:0000AD18         MOV      R3, #0x5F
.text:0000AD1C         STR      R3, [R11,#unk_constant]
.text:0000AD20         B        jump_to_write_power
.text:0000AD24 ; ---------------------------------------------------------
.text:0000AD24
.text:0000AD24 loc_AD24                 ; CODE XREF: main+6C0
.text:0000AD24         LDR      R2, [R11,#third_argument]
```

```
.text:0000AD28         MOV      R3, #899
.text:0000AD2C         CMP      R2, R3
.text:0000AD30         BGT      loc_AD40
.text:0000AD34         MOV      R3, #0x5A
.text:0000AD38         STR      R3, [R11,#unk_constant]
.text:0000AD3C         B        jump_to_write_power
.text:0000AD40 ; ------------------------------------------------------------
.text:0000AD40
.text:0000AD40 loc_AD40                  ; CODE XREF: main+6DC
.text:0000AD40         LDR      R2, [R11,#third_argument]
.text:0000AD44         MOV      R3, #999
.text:0000AD48         CMP      R2, R3
.text:0000AD4C         BGT      loc_AD5C
.text:0000AD50         MOV      R3, #0x55
.text:0000AD54         STR      R3, [R11,#unk_constant]
.text:0000AD58         B        jump_to_write_power
.text:0000AD5C ; ------------------------------------------------------------
.text:0000AD5C
.text:0000AD5C loc_AD5C                  ; CODE XREF: main+6F8
.text:0000AD5C         LDR      R2, [R11,#third_argument]
.text:0000AD60         MOV      R3, #1099
.text:0000AD64         CMP      R2, R3
.text:0000AD68         BGT      jump_to_write_power
.text:0000AD6C         MOV      R3, #0x50
.text:0000AD70         STR      R3, [R11,#unk_constant]
.text:0000AD74
.text:0000AD74 jump_to_write_power                  ; CODE XREF: main+6B0
.text:0000AD74                                      ; main+6CC ...
.text:0000AD74         LDR      R3, [R11,#var_28]
.text:0000AD78         UXTB     R1, R3
.text:0000AD7C         LDR      R3, [R11,#var_2C]
.text:0000AD80         UXTB     R2, R3
.text:0000AD84         LDR      R3, [R11,#unk_constant]
.text:0000AD88         UXTB     R3, R3
.text:0000AD8C         LDR      R0, [R11,#third_argument]
.text:0000AD90         UXTH     R0, R0
.text:0000AD94         STR      R0, [SP,#0x44+var_44]
.text:0000AD98         LDR      R0, [R11,#var_24]
.text:0000AD9C         BL       write_power
.text:0000ADA0         LDR      R0, [R11,#var_24]
.text:0000ADA4         MOV      R1, #0x5A
.text:0000ADA8         BL       read_loop
.text:0000ADAC         B        loc_ADD4

...

.rodata:0000B378 aSErrorWithArgu DCB "%s: Error with arguments",0xA,0 ; DATA
    XREF: main+684

...
```

Les noms de fonctions étaient présents dans les informations de débogage du binaire original, comme write_power, read_loop. Mais j'ai nommé les labels à l'intérieur

des fonctions.

Le nom optind semble familier. Il provient de la bibliothèque *NIX *getopt* qui sert à traiter les arguments de la ligne de commande—bien, c'est exactement ce qui se passe dans ce code. Ensuite, le 3ème argument (où la valeur de la fréquence est passée) est converti d'une chaîne vers un nombre en utilisant un appel à la fonction *strtoll()*.

La valeur est ensuite encore comparée par rapport à diverses constantes. En 0xACEC, elle est testée, si elle est inférieure ou égale à 499, et si c'est le cas, 0x64 est passé à la fonction write_power() (qui envoie une commande par USB en utilisant send_msg()). Si elle est plus grande que 499, un saut en 0xAD08 se produit.

En 0xAD08 on teste si elle est inférieure ou égale à 799. En cas de succès 0x5F est alors passé à la fonction write_power().

Il y a d'autres tests: par rapport à 899 en 0xAD24, à 0x999 en 0xAD40 et enfin, à 1099 en 0xAD5C. Si la fréquence est inférieure ou égale à 1099, 0x50 est passé (en 0xAD6C) à la fonction write_power(). Et il y a une sorte de bug. Si la valeur est encore plus grande que 1099, la valeur elle-même est passée à la fonction write_power(). Oh, ce n'est pas un bug, car nous ne pouvons pas arriver là: la valeur est d'abord comparée à 950 en 0xAC88, et si elle est plus grande, un message d'erreur est affiché et l'utilitaire s'arrête.

Maintenant, la table des fréquences en MHz et la valeur passée à la fonction write_power() :

MHz	héxadecimal	décimal
499MHz	0x64	100
799MHz	0x5f	95
899MHz	0x5a	90
999MHz	0x55	85
1099MHz	0x50	80

Il semble que la valeur passée à la carte décroît lorsque la fréquence croît.

Maintenant, nous voyons que la valeur de 950MHz est codée en dur, au moins dans cet utilitaire. Pouvons-nous le truquer?

Retournons à ce morceau de code:

```
.text:0000AC84      LDR     R2, [R11,#third_argument]
.text:0000AC88      MOV     R3, #950
.text:0000AC8C      CMP     R2, R3
.text:0000AC90      BGT     errors_with_arguments ; j'ai modifié ici en 00 00
    00 00
```

Nous devons désactiver l'instruction de branchement BGT en 0xAC90. Et ceci est du ARM en mode ARM, car, comme on le voit, toutes les adresses augmentent par 4, i.e, chaque instruction a une taille de 4 octets. L'instruction NOP (no operation) en mode ARM est juste quatre octets à zéro: 00 00 00 00. Donc en écrivant quatre octets à zéro à l'adresse 0xAC90 (ou à l'offset 0x2C90 dans le fichier), nous pouvons désactiver le test.

Maintenant, il est possible de définir la fréquence jusqu'à 1050MHz. Et même plus, mais, à cause du bug, si la valeur en entrée est plus grande que 1099, la valeur *telle quelle* en MHz sera passée à la carte, ce qui est incorrect.

Je ne suis pas allé plus loin, mais si je devais, j'essayerai de diminuer la valeur qui est passée à la fonction `write_power()`.

Maintenant, le morceau de code effrayant que j'ai passé en premier:

```
.text:0000AC94       LDR      R2, [R11,#third_argument]
.text:0000AC98       MOV      R3, #0x51EB851F
.text:0000ACA0       SMULL    R1, R3, R3, R2 ; R3=3rg_arg/3.125
.text:0000ACA4       MOV      R1, R3,ASR#4 ; R1=R3/16=3rg_arg/50
.text:0000ACA8       MOV      R3, R2,ASR#31 ; R3=MSB(3rg_arg)
.text:0000ACAC       RSB      R3, R3, R1 ; R3=3rd_arg/50
.text:0000ACB0       MOV      R1, #50
.text:0000ACB4       MUL      R3, R1, R3 ; R3=50*(3rd_arg/50)
.text:0000ACB8       RSB      R3, R3, R2
.text:0000ACBC       CMP      R3, #0
.text:0000ACC0       BEQ      loc_ACEC
.text:0000ACC4
.text:0000ACC4 errors_with_arguments
```

La division via la multiplication est utilisée ici, et la constante est 0x51EB851F. Je me suis écrit un petit calculateur pour programmeur[26]. Et il est capable de calculer le modulo inverse.

```
modinv32(0x51EB851F)
Warning, result is not integer: 3.125000
(unsigned) dec: 3 hex: 0x3 bin: 11
```

Cela signifie que l'instruction SMULL en 0xACA0 divise le 3ème argument par 3.125. En fait, tout ce que la fonction `modinv32()` de mon calculateur fait est ceci:

$$\frac{1}{\frac{input}{2^{32}}} = \frac{2^{32}}{input}$$

Ensuite il y a es décalages additionnels et maintenant nous voyons que le 3ème argument est simplement divisé par 50. Et ensuite il à nouveau multiplié par 50. Pourquoi? Ceci est un simple test, pour savoir si la valeur entrée est divisible par 50. Si la valeur de cette expression est non nulle, x n'est pas divisible par 50:

$$x - ((\frac{x}{50}) \cdot 50)$$

Ceci est en fait une manière simple de calculer le reste de la division.

Et alors, si le reste est non nul, un message d'erreur est affiché. Donc cet utilitaire prend des fréquences comme 850, 900, 950, 1000, etc., mais pas 855 ou 911.

[26] https://yurichev.com/progcalc/

C'est ça! Si vous faites quelque chose comme ça, soyez avertis que vous pouvez endommager votre carte, tout comme en cas d'overclocking d'autres éléments comme les CPUs, GPU[27]s, etc. Si vous avez une carte Cointerra, faites ceci à votre propre risque!

8.11 Casser un simple exécutable de chiffrement de code

Nous avons un fichier exécutable qui est chiffré par du chiffrement relativement simple. Il est ici (seule la section exécutable est laissée ici).

Tout d'abord, tout ce que fait la fonction de chiffrement, c'est d'ajouter l'index de la position dans le buffer à l'octet. Voici comment ça peut être implémenté:

Listing 8.9 : Python script

```
#!/usr/bin/env python
def e(i, k):
    return chr ((ord(i)+k) % 256)

def encrypt(buf):
    return e(buf[0], 0)+ e(buf[1], 1)+ e(buf[2], 2) + e(buf[3], 3)+ e(buf↙
    ↳ [4], 4)+ e(buf[5], 5)+ e(buf[6], 6)+ e(buf[7], 7)+
           e(buf[8], 8)+ e(buf[9], 9)+ e(buf[10], 10)+ e(buf[11], 11)+ e(↙
    ↳ buf[12], 12)+ e(buf[13], 13)+ e(buf[14], 14)+ e(buf[15], 15)
```

Ainsi, si vous chiffrez un buffer avec 16 zéros, vous obtiendrez *0, 1, 2, 3 ... 12, 13, 14, 15*.

La Propagating Cipher Block Chaining (PCBC) est aussi utilisée, voici comment elle fonctionne:

[27] Graphics Processing Unit

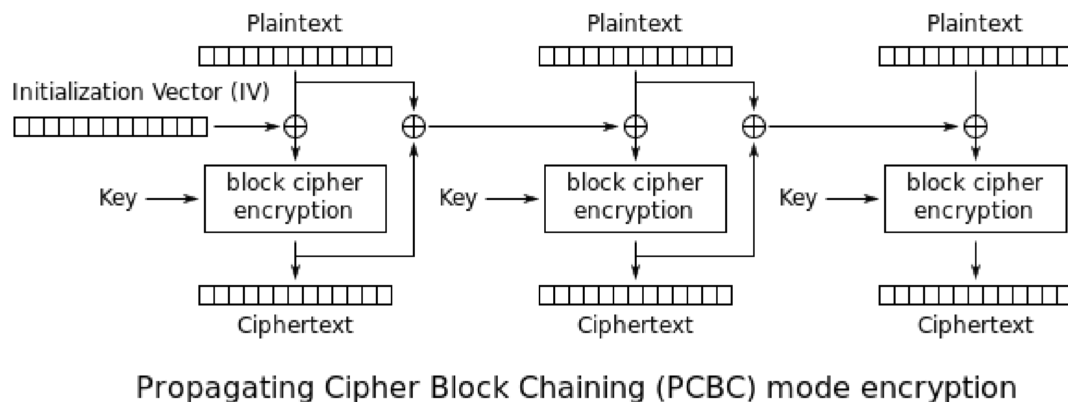


Fig. 8.15: Chiffrement avec Propagating Cipher Block Chaining (l'image provient d'un article Wikipédia)

Le problème est de retrouver l'IV. La force brute n'est pas une option, car l'IV est trop long (16 octets). Voyons s'il est possible de recouvrer l'IV pour un fichier binaire exécutable arbitraire?

Essayons la simple analyse de fréquence. Ceci est du code exécutable 32-bit x86, donc collectons des statistiques sur les octets et les opcodes les plus fréquents. J'ai essayé le fichier géant oracle.exe d' Oracle RDBMS version 11.2 pour windows x86 et j'ai trouvé que l'octet le plus fréquent (pas de surprise) est zéro (10%). L'octet suivant le plus fréquent est (encore une fois, sans surprise) 0xFF (5%). Le suivant est 0x8B (5%).

0x8B est l'opcode de MOV, ceci est en effet l'une des instructions x86 les plus fréquentes. Maintenant, que dire de la popularité de l'octet zéro? Si le compilateur doit encoder une valeur plus grande que 127, il doit utiliser un déplacement 32-bit au lieu d'un de 8-bit, mais les grandes valeurs sont très rares (2.1.8 on page 584), donc il est complété par des zéros. C'est le cas au moins avec LEA, MOV, PUSH, CALL.

Par exemple:

```
8D B0 28 01 00 00                  lea     esi, [eax+128h]
8D BF 40 38 00 00                  lea     edi, [edi+3840h]
```

Les déplacements plus grand que 127 sont très fréquents, mais ils excèdent rarement 0x10000 (en effet, des buffers mémoire/structures aussi grands sont aussi rares).

Même chose avec MOV, les grandes constantes sont rares, les plus utilisées sont 0, 1, 10, 100, 2^n, et ainsi de suite. Le compilateur doit compléter les petites constantes avec des zéros pour les encoder comme des valeurs 32-bit:

```
BF 02 00 00 00                     mov     edi, 2
BF 01 00 00 00                     mov     edi, 1
```

Maintenant parlons des octets 00 et 0xFF combinés: les sauts (conditionnels inclus) et appels peuvent transférer le flux d'exécution en avant ou en arrière, mais très souvent, dans les limites du module exécutable courant. Si c'est en avant, le déplacement n'est pas très grand et il y a des zéros ajoutés. Si c'est en arrière, le déplacement est représenté par une valeur négative, donc complétée par des octets 0xFF. Par exemple, transfert du flux d'exécution en avant:

```
E8 43 0C 00 00                  call    _function1
E8 5C 00 00 00                  call    _function2
0F 84 F0 0A 00 00               jz      loc_4F09A0
0F 84 EB 00 00 00               jz      loc_4EFBB8
```

En arrière:

```
E8 79 0C FE FF                  call    _function1
E8 F4 16 FF FF                  call    _function2
0F 84 F8 FB FF FF               jz      loc_8212BC
0F 84 06 FD FF FF               jz      loc_FF1E7D
```

L'octet 0xFF se rencontre aussi très souvent dans des déplacements négatifs, comme ceux-ci:

```
8D 85 1E FF FF FF               lea     eax, [ebp-0E2h]
8D 95 F8 5C FF FF               lea     edx, [ebp-0A308h]
```

Jusqu'ici, tout va bien. Maintenant nous devons essayer diverses clefs 16-octet, déchiffrer la section exécutable et mesurer les occurences des octets 0, 0xFF et 0x8B. Gardons en vue la façon dont le déchiffrement PCBC fonctionne:

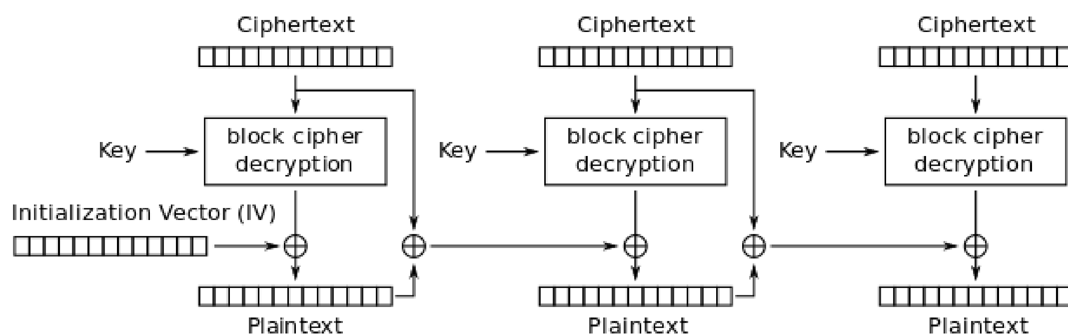


Fig. 8.16: Propagating Cipher Block Chaining decryption (l'image provient d'un article Wikipédia)

La bonne nouvelle est que nous n'avons pas vraiment besoin de déchiffrer l'ensemble des données, mais seulement slice par slice, ceci est exactement comment j'ai procédé dans mon exemple précédent: 9.1.5 on page 1218.

Maintenant j'essaye tous les octets possible (0..255) pour chaque octet dans la clef et je prends l'octet produisant le plus grande nombre d'octets 0x0/0xFF/0x8B dans le slice déchiffré:

```
#!/usr/bin/env python
import sys, hexdump, array, string, operator

KEY_LEN=16

def chunks(l, n):
    # split n by l-byte chunks
    # https://stackoverflow.com/q/312443
    n = max(1, n)
    return [l[i:i + n] for i in range(0, len(l), n)]

def read_file(fname):
    file=open(fname, mode='rb')
    content=file.read()
    file.close()
    return content

def decrypt_byte (c, key):
    return chr((ord(c)-key) % 256)

def XOR_PCBC_step (IV, buf, k):
    prev=IV
    rt=""
    for c in buf:
        new_c=decrypt_byte(c, k)
        plain=chr(ord(new_c)^ord(prev))
        prev=chr(ord(c)^ord(plain))
        rt=rt+plain
    return rt

each_Nth_byte=[""]*KEY_LEN

content=read_file(sys.argv[1])
# split input by 16-byte chunks:
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks:
    for i in range(KEY_LEN):
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN):
    print "N=", N
    stat={}
    for i in range(256):
        tmp_key=chr(i)
        tmp=XOR_PCBC_step(tmp_key,each_Nth_byte[N], N)
        # count 0, FFs and 8Bs in decrypted buffer:
        important_bytes=tmp.count('\x00')+tmp.count('\xFF')+tmp.count('\x8B↙
    ↳ ')
        stat[i]=important_bytes
```

```
    sorted_stat = sorted(stat.iteritems(), key=operator.itemgetter(1), ↙
    ↳ reverse=True)
    print sorted_stat[0]
```

(Le code source peut être téléchargé ici.)

Je le lance et voici une clef pour laquelle le nombre d'octets 0/0xFF/0x8B dans le buffer déchiffré est maximum:

```
N= 0
(147, 1224)
N= 1
(94, 1327)
N= 2
(252, 1223)
N= 3
(218, 1266)
N= 4
(38, 1209)
N= 5
(192, 1378)
N= 6
(199, 1204)
N= 7
(213, 1332)
N= 8
(225, 1251)
N= 9
(112, 1223)
N= 10
(143, 1177)
N= 11
(108, 1286)
N= 12
(10, 1164)
N= 13
(3, 1271)
N= 14
(128, 1253)
N= 15
(232, 1330)
```

Écrivons un utilitaire de déchiffrement avec la clef obtenue:

```
#!/usr/bin/env python
import sys, hexdump, array

def xor_strings(s,t):
    # https://en.wikipedia.org/wiki/XOR_cipher#Example_implementation
    """xor two strings together"""
    return "".join(chr(ord(a)^ord(b)) for a,b in zip(s,t))

IV=array.array('B', [147, 94, 252, 218, 38, 192, 199, 213, 225, 112, 143, ↙
    ↳ 108, 10, 3, 128, 232]).tostring()
```

```
def chunks(l, n):
    n = max(1, n)
    return [l[i:i + n] for i in range(0, len(l), n)]

def read_file(fname):
    file=open(fname, mode='rb')
    content=file.read()
    file.close()
    return content

def decrypt_byte(i, k):
    return chr ((ord(i)-k) % 256)

def decrypt(buf):
    return "".join(decrypt_byte(buf[i], i) for i in range(16))

fout=open(sys.argv[2], mode='wb')

prev=IV
content=read_file(sys.argv[1])
tmp=chunks(content, 16)
for c in tmp:
    new_c=decrypt(c)
    p=xor_strings (new_c, prev)
    prev=xor_strings(c, p)
    fout.write(p)
fout.close()
```

(Le code source peut être téléchargé ici.)

Vérifions le fichier résultant:

```
$ objdump -b binary -m i386 -D decrypted.bin

...

       5:       8b ff                   mov    %edi,%edi
       7:       55                      push   %ebp
       8:       8b ec                   mov    %esp,%ebp
       a:       51                      push   %ecx
       b:       53                      push   %ebx
       c:       33 db                   xor    %ebx,%ebx
       e:       43                      inc    %ebx
       f:       84 1d a0 e2 05 01       test   %bl,0x105e2a0
      15:       75 09                   jne    0x20
      17:       ff 75 08                pushl  0x8(%ebp)
      1a:       ff 15 b0 13 00 01       call   *0x10013b0
      20:       6a 6c                   push   $0x6c
      22:       ff 35 54 d0 01 01       pushl  0x101d054
      28:       ff 15 b4 13 00 01       call   *0x10013b4
      2e:       89 45 fc                mov    %eax,-0x4(%ebp)
      31:       85 c0                   test   %eax,%eax
      33:       0f 84 d9 00 00 00       je     0x112
```

```
      39:       56                      push   %esi
      3a:       57                      push   %edi
      3b:       6a 00                   push   $0x0
      3d:       50                      push   %eax
      3e:       ff 15 b8 13 00 01       call   *0x10013b8
      44:       8b 35 bc 13 00 01       mov    0x10013bc,%esi
      4a:       8b f8                   mov    %eax,%edi
      4c:       a1 e0 e2 05 01          mov    0x105e2e0,%eax
      51:       3b 05 e4 e2 05 01       cmp    0x105e2e4,%eax
      57:       75 12                   jne    0x6b
      59:       53                      push   %ebx
      5a:       6a 03                   push   $0x3
      5c:       57                      push   %edi
      5d:       ff d6                   call   *%esi

...
```

Oui, ceci semble être un morceau correctement désassemblé de code x86. Le fichier déchiffré entier peut être téléchargé ici.

En fait, ceci est la section text du regedit.exe de Windows 7. Mais cet exemple est basé sur un cas réel que j'ai rencontré, seul l'exécutable est différent (et la clef), l'algorithme est le même.

8.11.1 Autres idées à prendre en considération

Et si j'avais échoué avec cette simple analyse des fréquences? Il y a d'autres idées sur la façon de mesurer l'exactitude de code x86 déchiffré/décompressé:

- De nombreux compilateurs modernes alignent le début des fonctions sur une limite 16-bits. Donc l'espace libre avant est rempli avec de NOPs (0x90) ou d'autres instructions avec des opcodes connus: .1.7 on page 1347. Ou des instructions INT3 (0xCC).
- Peut-être que le pattern le plus fréquent dans tout langage d'assemblage est l'appel de fonction:
  `PUSH chain / CALL / ADD ESP, X`. Cette séquence peut facilement être détectée et trouvée. J'ai même collecté des statistiques sur le nombre moyen d'arguments des fonctions: 11.3 on page 1289. (Ainsi, ceci est la longueur moyenne d'une série d'instructions PUSH.)

En savoir plus sur le code désassemblé incorrectement/correctement: 5.11 on page 944.

8.12 SAP

8.12.1 À propos de la compression du trafic réseau par le client SAP

(Tracer la connexion entre la variable d'environnement TDW_NOCOMPRESS SAPGUI[28] et la fenêtre pop-up gênante et ennuyeuse et la routine de compression de données actuelle.)

On sait que le trafic réseau entre le SAPGUI et SAP n'est pas chiffré par défaut, mais compressé (voir ici[29] et ici[30]).

Il est aussi connue que mettre la variable d'environnement *TDW_NOCOMPRESS* à 1, permet d'arrêter la compression des paquets réseau.

Mais vous verrez toujours l'ennuyeuse fenêtre pop-up, qui ne peut pas être fermée:

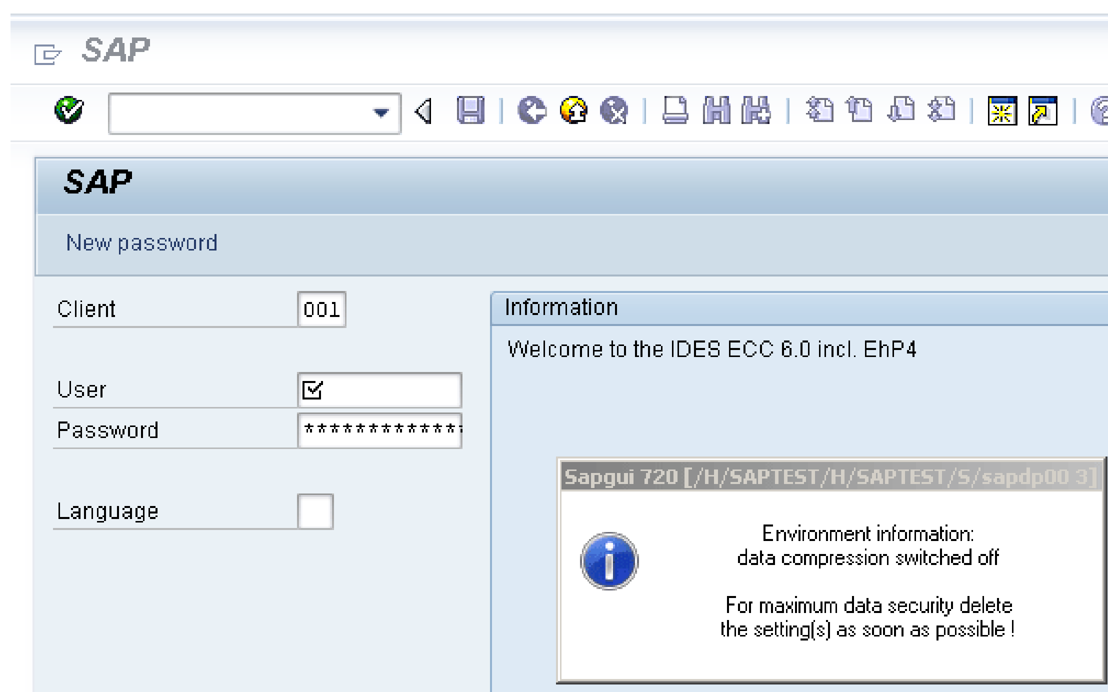


Fig. 8.17: Screenshot

Voyons si nous pouvons supprimer cette fenêtre.

Mais avant, voyons ce que nous savons déjà:

[28]client SAP GUI
[29]http://blog.yurichev.com/node/44
[30]blog.yurichev.com

Premièrement: nous savons que la variable d'environnement *TDW_NOCOMPRESS* est vérifiée quelque part dans le client SAPGUI.

Deuxièmement: une chaîne comme «data compression switched off » doit s'y trouver quelque part.

Avec l'aide du gestionnaire de fichier FAR[31]nous pouvons trouver que deux de ces chaînes sont stockées dans le fichier SAPguilib.dll.

Donc ouvrons SAPguilib.dll dans IDA et cherchons la chaîne *TDW_NOCOMPRESS*. Oui, elle s'y trouve et il n'y a qu'une référence vers elle.

Nous voyons le morceau de code suivant (tous les offsets de fichiers sont valables pour SAPGUI 720 win32, fichier SAPguilib.dll version 7200,1,0,9009) :

```
.text:6440D51B                lea     eax, [ebp+2108h+var_211C]
.text:6440D51E                push    eax             ; int
.text:6440D51F                push    offset aTdw_nocompress ;
    "TDW_NOCOMPRESS"
.text:6440D524                mov     byte ptr [edi+15h], 0
.text:6440D528                call    chk_env
.text:6440D52D                pop     ecx
.text:6440D52E                pop     ecx
.text:6440D52F                push    offset byte_64443AF8
.text:6440D534                lea     ecx, [ebp+2108h+var_211C]

; demangled name: int ATL::CStringT::Compare(char const *)const
.text:6440D537                call    ds:mfc90_1603
.text:6440D53D                test    eax, eax
.text:6440D53F                jz      short loc_6440D55A
.text:6440D541                lea     ecx, [ebp+2108h+var_211C]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:6440D544                call    ds:mfc90_910
.text:6440D54A                push    eax             ; Str
.text:6440D54B                call    ds:atoi
.text:6440D551                test    eax, eax
.text:6440D553                setnz   al
.text:6440D556                pop     ecx
.text:6440D557                mov     [edi+15h], al
```

La chaîne renvoyée par chk_env() via son second argument est ensuite traitée par la fonction de chaîne MFC et ensuite `atoi()`[32] est appelée. Après ça, la valeur numérique est stockée en edi+15h.

Jetons aussi un œil à la fonction chk_env() (nous avons donné ce nom manuellement) :

```
.text:64413F20 ; int __cdecl chk_env(char *VarName, int)
.text:64413F20 chk_env         proc near
.text:64413F20
.text:64413F20 DstSize         = dword ptr -0Ch
```

[31] http://www.farmanager.com/
[32] fonction C standard qui convertit les chiffres d'une chaîne en un nombre

```
.text:64413F20 var_8           = dword ptr -8
.text:64413F20 DstBuf          = dword ptr -4
.text:64413F20 VarName         = dword ptr  8
.text:64413F20 arg_4           = dword ptr  0Ch
.text:64413F20
.text:64413F20                 push    ebp
.text:64413F21                 mov     ebp, esp
.text:64413F23                 sub     esp, 0Ch
.text:64413F26                 mov     [ebp+DstSize], 0
.text:64413F2D                 mov     [ebp+DstBuf], 0
.text:64413F34                 push    offset unk_6444C88C
.text:64413F39                 mov     ecx, [ebp+arg_4]

; (demangled name) ATL::CStringT::operator=(char const *)
.text:64413F3C                 call    ds:mfc90_820
.text:64413F42                 mov     eax, [ebp+VarName]
.text:64413F45                 push    eax             ; VarName
.text:64413F46                 mov     ecx, [ebp+DstSize]
.text:64413F49                 push    ecx             ; DstSize
.text:64413F4A                 mov     edx, [ebp+DstBuf]
.text:64413F4D                 push    edx             ; DstBuf
.text:64413F4E                 lea     eax, [ebp+DstSize]
.text:64413F51                 push    eax             ; ReturnSize
.text:64413F52                 call    ds:getenv_s
.text:64413F58                 add     esp, 10h
.text:64413F5B                 mov     [ebp+var_8], eax
.text:64413F5E                 cmp     [ebp+var_8], 0
.text:64413F62                 jz      short loc_64413F68
.text:64413F64                 xor     eax, eax
.text:64413F66                 jmp     short loc_64413FBC
.text:64413F68
.text:64413F68 loc_64413F68:
.text:64413F68                 cmp     [ebp+DstSize], 0
.text:64413F6C                 jnz     short loc_64413F72
.text:64413F6E                 xor     eax, eax
.text:64413F70                 jmp     short loc_64413FBC
.text:64413F72
.text:64413F72 loc_64413F72:
.text:64413F72                 mov     ecx, [ebp+DstSize]
.text:64413F75                 push    ecx
.text:64413F76                 mov     ecx, [ebp+arg_4]

; demangled name: ATL::CSimpleStringT<char, 1>::Preallocate(int)
.text:64413F79                 call    ds:mfc90_2691
.text:64413F7F                 mov     [ebp+DstBuf], eax
.text:64413F82                 mov     edx, [ebp+VarName]
.text:64413F85                 push    edx             ; VarName
.text:64413F86                 mov     eax, [ebp+DstSize]
.text:64413F89                 push    eax             ; DstSize
.text:64413F8A                 mov     ecx, [ebp+DstBuf]
.text:64413F8D                 push    ecx             ; DstBuf
.text:64413F8E                 lea     edx, [ebp+DstSize]
.text:64413F91                 push    edx             ; ReturnSize
```

```
.text:64413F92                  call    ds:getenv_s
.text:64413F98                  add     esp, 10h
.text:64413F9B                  mov     [ebp+var_8], eax
.text:64413F9E                  push    0FFFFFFFFh
.text:64413FA0                  mov     ecx, [ebp+arg_4]

; demangled name: ATL::CSimpleStringT::ReleaseBuffer(int)
.text:64413FA3                  call    ds:mfc90_5835
.text:64413FA9                  cmp     [ebp+var_8], 0
.text:64413FAD                  jz      short loc_64413FB3
.text:64413FAF                  xor     eax, eax
.text:64413FB1                  jmp     short loc_64413FBC
.text:64413FB3
.text:64413FB3 loc_64413FB3:
.text:64413FB3                  mov     ecx, [ebp+arg_4]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64413FB6                  call    ds:mfc90_910
.text:64413FBC
.text:64413FBC loc_64413FBC:
.text:64413FBC
.text:64413FBC                  mov     esp, ebp
.text:64413FBE                  pop     ebp
.text:64413FBF                  retn
.text:64413FBF chk_env          endp
```

Oui. La fonction `getenv_s()`[33]

est une version de Microsoft à la sécurité avancée de `getenv()`[34].

Il y a quelques manipulation de chaîne MFC.

De nombreuses autres variables d'environnement sont également testées. Voici une liste de toutes les variables qui sont testé et ce que SAPGUI écrirait dans son fichier de log, lorsque les traces sont activées:

[33] MSDN
[34] Fonction de la bibliothèque C standard renvoyant une variable d'environnement

DPTRACE	“GUI-OPTION: Trace set to %d”
TDW_HEXDUMP	“GUI-OPTION: Hexdump enabled”
TDW_WORKDIR	“GUI-OPTION: working directory ‘%s’’
TDW_SPLASHSRCEENOFF	“GUI-OPTION: Splash Screen Off”
	“GUI-OPTION: Splash Screen On”
TDW_REPLYTIMEOUT	“GUI-OPTION: reply timeout %d milliseconds”
TDW_PLAYBACKTIMEOUT	“GUI-OPTION: PlaybackTimeout set to %d milliseconds”
TDW_NOCOMPRESS	“GUI-OPTION: no compression read”
TDW_EXPERT	“GUI-OPTION: expert mode”
TDW_PLAYBACKPROGRESS	“GUI-OPTION: PlaybackProgress”
TDW_PLAYBACKNETTRAFFIC	“GUI-OPTION: PlaybackNetTraffic”
TDW_PLAYLOG	“GUI-OPTION: /PlayLog is YES, file %s”
TDW_PLAYTIME	“GUI-OPTION: /PlayTime set to %d milliseconds”
TDW_LOGFILE	“GUI-OPTION: TDW_LOGFILE ‘%s’’
TDW_WAN	“GUI-OPTION: WAN - low speed connection enabled”
TDW_FULLMENU	“GUI-OPTION: FullMenu enabled”
SAP_CP / SAP_CODEPAGE	“GUI-OPTION: SAP_CODEPAGE ‘%d’’
UPDOWNLOAD_CP	“GUI-OPTION: UPDOWNLOAD_CP ‘%d’’
SNC_PARTNERNAME	“GUI-OPTION: SNC name ‘%s’’
SNC_QOP	“GUI-OPTION: SNC_QOP ‘%s’’
SNC_LIB	“GUI-OPTION: SNC is set to: %s”
SAPGUI_INPLACE	“GUI-OPTION: environment variable SAPGUI_INPLACE is on”

La configuration de chaque variable est écrit dans le tableau via le pointeur dans le registre EDI. EDI est renseigné avant l’appel à la fonction:

```
.text:6440EE00              lea     edi, [ebp+2884h+var_2884] ; options
    here like +0x15...
.text:6440EE03              lea     ecx, [esi+24h]
.text:6440EE06              call    load_command_line
.text:6440EE0B              mov     edi, eax
.text:6440EE0D              xor     ebx, ebx
.text:6440EE0F              cmp     edi, ebx
.text:6440EE11              jz      short loc_6440EE42
.text:6440EE13              push    edi
.text:6440EE14              push    offset aSapguiStoppedA ; "Sapgui
    stopped after commandline interp"...
.text:6440EE19              push    dword_644F93E8
.text:6440EE1F              call    FEWTraceError
```

Maintenant, pouvons-nous trouver la chaîne *data record mode switched on* ?

Oui, et la seule référence est dans

CDwsGui::PrepareInfoWindow().

Comment connaissons-nous les noms de classe/méthode? Il y a beaucoup d’appels spéciaux de débogage qui écrivent dans les fichiers de log, comme:

```
.text:64405160              push    dword ptr [esi+2854h]
.text:64405166              push    offset aCdwsguiPrepare ;
    "\nCDwsGui::PrepareInfoWindow: sapgui env"...
.text:6440516B              push    dword ptr [esi+2848h]
.text:64405171              call    dbg
.text:64405176              add     esp, 0Ch
```

...ou:

```
.text:6440237A                 push    eax
.text:6440237B                 push    offset aCclientStart_6 ;
    "CClient::Start: set shortcut user to '%'"...
.text:64402380                 push    dword ptr [edi+4]
.text:64402383                 call    dbg
.text:64402388                 add     esp, 0Ch
```

C’est *très* utile.

Voyons le contenu de la fonction de cette fenêtre pop-up ennuyeuse:

```
.text:64404F4F CDwsGui__PrepareInfoWindow proc near
.text:64404F4F
.text:64404F4F pvParam         = byte ptr -3Ch
.text:64404F4F var_38          = dword ptr -38h
.text:64404F4F var_34          = dword ptr -34h
.text:64404F4F rc              = tagRECT ptr -2Ch
.text:64404F4F cy              = dword ptr -1Ch
.text:64404F4F h               = dword ptr -18h
.text:64404F4F var_14          = dword ptr -14h
.text:64404F4F var_10          = dword ptr -10h
.text:64404F4F var_4           = dword ptr -4
.text:64404F4F
.text:64404F4F                 push    30h
.text:64404F51                 mov     eax, offset loc_64438E00
.text:64404F56                 call    __EH_prolog3
.text:64404F5B                 mov     esi, ecx        ; ECX is pointer to
    object
.text:64404F5D                 xor     ebx, ebx
.text:64404F5F                 lea     ecx, [ebp+var_14]
.text:64404F62                 mov     [ebp+var_10], ebx

; demangled name: ATL::CStringT(void)
.text:64404F65                 call    ds:mfc90_316
.text:64404F6B                 mov     [ebp+var_4], ebx
.text:64404F6E                 lea     edi, [esi+2854h]
.text:64404F74                 push    offset aEnvironmentInf ;
    "Environment information:\n"
.text:64404F79                 mov     ecx, edi

; demangled name: ATL::CStringT::operator=(char const *)
.text:64404F7B                 call    ds:mfc90_820
.text:64404F81                 cmp     [esi+38h], ebx
.text:64404F84                 mov     ebx, ds:mfc90_2539
.text:64404F8A                 jbe     short loc_64404FA9
.text:64404F8C                 push    dword ptr [esi+34h]
.text:64404F8F                 lea     eax, [ebp+var_14]
.text:64404F92                 push    offset aWorkingDirecto ;
    "working directory: '%s'\n"
.text:64404F97                 push    eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404F98                 call    ebx ; mfc90_2539
```

```
.text:64404F9A                 add     esp, 0Ch
.text:64404F9D                 lea     eax, [ebp+var_14]
.text:64404FA0                 push    eax
.text:64404FA1                 mov     ecx, edi

;
    demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)
.text:64404FA3                 call    ds:mfc90_941
.text:64404FA9
.text:64404FA9 loc_64404FA9:
.text:64404FA9                 mov     eax, [esi+38h]
.text:64404FAC                 test    eax, eax
.text:64404FAE                 jbe     short loc_64404FD3
.text:64404FB0                 push    eax
.text:64404FB1                 lea     eax, [ebp+var_14]
.text:64404FB4                 push    offset aTraceLevelDAct ;
    "trace level %d activated\n"
.text:64404FB9                 push    eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404FBA                 call    ebx ; mfc90_2539
.text:64404FBC                 add     esp, 0Ch
.text:64404FBF                 lea     eax, [ebp+var_14]
.text:64404FC2                 push    eax
.text:64404FC3                 mov     ecx, edi

;
    demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)
.text:64404FC5                 call    ds:mfc90_941
.text:64404FCB                 xor     ebx, ebx
.text:64404FCD                 inc     ebx
.text:64404FCE                 mov     [ebp+var_10], ebx
.text:64404FD1                 jmp     short loc_64404FD6
.text:64404FD3
.text:64404FD3 loc_64404FD3:
.text:64404FD3                 xor     ebx, ebx
.text:64404FD5                 inc     ebx
.text:64404FD6
.text:64404FD6 loc_64404FD6:
.text:64404FD6                 cmp     [esi+38h], ebx
.text:64404FD9                 jbe     short loc_64404FF1
.text:64404FDB                 cmp     dword ptr [esi+2978h], 0
.text:64404FE2                 jz      short loc_64404FF1
.text:64404FE4                 push    offset aHexdumpInTrace ;
    "hexdump in trace activated\n"
.text:64404FE9                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FEB                 call    ds:mfc90_945
.text:64404FF1
.text:64404FF1 loc_64404FF1:
.text:64404FF1
.text:64404FF1                 cmp     byte ptr [esi+78h], 0
.text:64404FF5                 jz      short loc_64405007
```

```
.text:64404FF7                 push    offset aLoggingActivat ;
    "logging activated\n"
.text:64404FFC                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FFE                 call    ds:mfc90_945
.text:64405004                 mov     [ebp+var_10], ebx
.text:64405007
.text:64405007 loc_64405007:
.text:64405007                 cmp     byte ptr [esi+3Dh], 0
.text:6440500B                 jz      short bypass
.text:6440500D                 push    offset aDataCompressio ;
    "data compression switched off\n"
.text:64405012                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014                 call    ds:mfc90_945
.text:6440501A                 mov     [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
.text:6440501D                 mov     eax, [esi+20h]
.text:64405020                 test    eax, eax
.text:64405022                 jz      short loc_6440503A
.text:64405024                 cmp     dword ptr [eax+28h], 0
.text:64405028                 jz      short loc_6440503A
.text:6440502A                 push    offset aDataRecordMode ;
    "data record mode switched on\n"
.text:6440502F                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405031                 call    ds:mfc90_945
.text:64405037                 mov     [ebp+var_10], ebx
.text:6440503A
.text:6440503A loc_6440503A:
.text:6440503A
.text:6440503A                 mov     ecx, edi
.text:6440503C                 cmp     [ebp+var_10], ebx
.text:6440503F                 jnz     loc_64405142
.text:64405045                 push    offset aForMaximumData ;
    "\nFor maximum data security delete\nthe s"...

; demangled name: ATL::CStringT::operator+=(char const *)
.text:6440504A                 call    ds:mfc90_945
.text:64405050                 xor     edi, edi
.text:64405052                 push    edi             ; fWinIni
.text:64405053                 lea     eax, [ebp+pvParam]
.text:64405056                 push    eax             ; pvParam
.text:64405057                 push    edi             ; uiParam
.text:64405058                 push    30h             ; uiAction
.text:6440505A                 call    ds:SystemParametersInfoA
.text:64405060                 mov     eax, [ebp+var_34]
.text:64405063                 cmp     eax, 1600
.text:64405068                 jle     short loc_64405072
.text:6440506A                 cdq
```

```
.text:6440506B                 sub     eax, edx
.text:6440506D                 sar     eax, 1
.text:6440506F                 mov     [ebp+var_34], eax
.text:64405072
.text:64405072 loc_64405072:
.text:64405072                 push    edi             ; hWnd
.text:64405073                 mov     [ebp+cy], 0A0h
.text:6440507A                 call    ds:GetDC
.text:64405080                 mov     [ebp+var_10], eax
.text:64405083                 mov     ebx, 12Ch
.text:64405088                 cmp     eax, edi
.text:6440508A                 jz      loc_64405113
.text:64405090                 push    11h             ; i
.text:64405092                 call    ds:GetStockObject
.text:64405098                 mov     edi, ds:SelectObject
.text:6440509E                 push    eax             ; h
.text:6440509F                 push    [ebp+var_10]    ; hdc
.text:644050A2                 call    edi ; SelectObject
.text:644050A4                 and     [ebp+rc.left], 0
.text:644050A8                 and     [ebp+rc.top], 0
.text:644050AC                 mov     [ebp+h], eax
.text:644050AF                 push    401h            ; format
.text:644050B4                 lea     eax, [ebp+rc]
.text:644050B7                 push    eax             ; lprc
.text:644050B8                 lea     ecx, [esi+2854h]
.text:644050BE                 mov     [ebp+rc.right], ebx
.text:644050C1                 mov     [ebp+rc.bottom], 0B4h

; demangled name: ATL::CSimpleStringT::GetLength(void)
.text:644050C8                 call    ds:mfc90_3178
.text:644050CE                 push    eax             ; cchText
.text:644050CF                 lea     ecx, [esi+2854h]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:644050D5                 call    ds:mfc90_910
.text:644050DB                 push    eax             ; lpchText
.text:644050DC                 push    [ebp+var_10]    ; hdc
.text:644050DF                 call    ds:DrawTextA
.text:644050E5                 push    4               ; nIndex
.text:644050E7                 call    ds:GetSystemMetrics
.text:644050ED                 mov     ecx, [ebp+rc.bottom]
.text:644050F0                 sub     ecx, [ebp+rc.top]
.text:644050F3                 cmp     [ebp+h], 0
.text:644050F7                 lea     eax, [eax+ecx+28h]
.text:644050FB                 mov     [ebp+cy], eax
.text:644050FE                 jz      short loc_64405108
.text:64405100                 push    [ebp+h]         ; h
.text:64405103                 push    [ebp+var_10]    ; hdc
.text:64405106                 call    edi ; SelectObject
.text:64405108
.text:64405108 loc_64405108:
.text:64405108                 push    [ebp+var_10]    ; hDC
.text:6440510B                 push    0               ; hWnd
```

```
.text:6440510D                 call    ds:ReleaseDC
.text:64405113
.text:64405113 loc_64405113:
.text:64405113                 mov     eax, [ebp+var_38]
.text:64405116                 push    80h             ; uFlags
.text:6440511B                 push    [ebp+cy]        ; cy
.text:6440511E                 inc     eax
.text:6440511F                 push    ebx             ; cx
.text:64405120                 push    eax             ; Y
.text:64405121                 mov     eax, [ebp+var_34]
.text:64405124                 add     eax, 0FFFFFED4h
.text:64405129                 cdq
.text:6440512A                 sub     eax, edx
.text:6440512C                 sar     eax, 1
.text:6440512E                 push    eax             ; X
.text:6440512F                 push    0               ; hWndInsertAfter
.text:64405131                 push    dword ptr [esi+285Ch] ; hWnd
.text:64405137                 call    ds:SetWindowPos
.text:6440513D                 xor     ebx, ebx
.text:6440513F                 inc     ebx
.text:64405140                 jmp     short loc_6440514D
.text:64405142
.text:64405142 loc_64405142:
.text:64405142                 push    offset byte_64443AF8

; demangled name: ATL::CStringT::operator=(char const *)
.text:64405147                 call    ds:mfc90_820
.text:6440514D
.text:6440514D loc_6440514D:
.text:6440514D                 cmp     dword_6450B970, ebx
.text:64405153                 jl      short loc_64405188
.text:64405155                 call    sub_6441C910
.text:6440515A                 mov     dword_644F858C, ebx
.text:64405160                 push    dword ptr [esi+2854h]
.text:64405166                 push    offset aCdwsguiPrepare ;
    "\nCDwsGui::PrepareInfoWindow: sapgui env"...
.text:6440516B                 push    dword ptr [esi+2848h]
.text:64405171                 call    dbg
.text:64405176                 add     esp, 0Ch
.text:64405179                 mov     dword_644F858C, 2
.text:64405183                 call    sub_6441C920
.text:64405188
.text:64405188 loc_64405188:
.text:64405188                 or      [ebp+var_4], 0FFFFFFFFh
.text:6440518C                 lea     ecx, [ebp+var_14]

; demangled name: ATL::CStringT:: CStringT()
.text:6440518F                 call    ds:mfc90_601
.text:64405195                 call    __EH_epilog3
.text:6440519A                 retn
.text:6440519A CDwsGui__PrepareInfoWindow endp
```

Au début de la fonction, ECX a un pointeur sur l’objet (puisque c’est une fonction avec

le type d'appel thiscall (3.21.1 on page 699)). Dans notre cas, l'objet a étonnement un type de classe de *CDwsGui*. En fonction de l'option mise dans l'objet, un message spécifique est concaténé au message résultant.

Si la valeur à l'adresse `this+0x3D` n'est pas zéro, la compression est désactivée:

```
.text:64405007 loc_64405007:
.text:64405007                 cmp     byte ptr [esi+3Dh], 0
.text:6440500B                 jz      short bypass
.text:6440500D                 push    offset aDataCompressio ;
    "data compression switched off\n"
.text:64405012                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014                 call    ds:mfc90_945
.text:6440501A                 mov     [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
```

Finalement, il est intéressant de noter que l'état de la variable *var_10* défini si le message est affiché:

```

.text:6440503C                 cmp     [ebp+var_10], ebx
.text:6440503F                 jnz     exit ; passe outre l'affichage

; ajoute les chaînes "For maximum data security delete" / "the setting(s) as
    soon as possible !":

.text:64405045                 push    offset aForMaximumData ;
    "\nFor maximum data security delete\nthe s"...
.text:6440504A                 call    ds:mfc90_945 ;
    ATL::CStringT::operator+=(char const *)
.text:64405050                 xor     edi, edi
.text:64405052                 push    edi             ; fWinIni
.text:64405053                 lea     eax, [ebp+pvParam]
.text:64405056                 push    eax             ; pvParam
.text:64405057                 push    edi             ; uiParam
.text:64405058                 push    30h             ; uiAction
.text:6440505A                 call    ds:SystemParametersInfoA
.text:64405060                 mov     eax, [ebp+var_34]
.text:64405063                 cmp     eax, 1600
.text:64405068                 jle     short loc_64405072
.text:6440506A                 cdq
.text:6440506B                 sub     eax, edx
.text:6440506D                 sar     eax, 1
.text:6440506F                 mov     [ebp+var_34], eax
.text:64405072
.text:64405072 loc_64405072:

start drawing:

.text:64405072                 push    edi             ; hWnd
.text:64405073                 mov     [ebp+cy], 0A0h
.text:6440507A                 call    ds:GetDC
```

Vérifions notre théorie en pratique.

JNZ à cette ligne …

```
.text:6440503F                jnz     exit ; passe outre l'affichage
```

…remplaçons-le par un JMP, et nous obtenons SAPGUI fonctionnant sans que l'ennuyeuse fenêtre pop-up n'apparaisse!

Maintenant approfondissons et trouvons la relation entre l'offset $0x15$ dans la fonction `load_command_line()` (nous lui avons donné ce nom) et la variable `this+0x3D` dans *CDwsGui::PrepareInfoWindow*. Sommes-nous sûrs que la valeur est la même?

Nous commençons par chercher toutes les occurrences de la valeur `0x15` dans le code. Pour un petit programme comme SAPGUI, cela fonctionne parfois.a Voici la première occurrence que nous obtenons:

```
.text:64404C19 sub_64404C19    proc near
.text:64404C19
.text:64404C19 arg_0           = dword ptr  4
.text:64404C19
.text:64404C19                 push    ebx
.text:64404C1A                 push    ebp
.text:64404C1B                 push    esi
.text:64404C1C                 push    edi
.text:64404C1D                 mov     edi, [esp+10h+arg_0]
.text:64404C21                 mov     eax, [edi]
.text:64404C23                 mov     esi, ecx ; ESI/ECX sont des pointeurs
    sur un objet inconnu
.text:64404C25                 mov     [esi], eax
.text:64404C27                 mov     eax, [edi+4]
.text:64404C2A                 mov     [esi+4], eax
.text:64404C2D                 mov     eax, [edi+8]
.text:64404C30                 mov     [esi+8], eax
.text:64404C33                 lea     eax, [edi+0Ch]
.text:64404C36                 push    eax
.text:64404C37                 lea     ecx, [esi+0Ch]

; demangled name:  ATL::CStringT::operator=(class ATL::CStringT ... &)
.text:64404C3A                 call    ds:mfc90_817
.text:64404C40                 mov     eax, [edi+10h]
.text:64404C43                 mov     [esi+10h], eax
.text:64404C46                 mov     al, [edi+14h]
.text:64404C49                 mov     [esi+14h], al
.text:64404C4C                 mov     al, [edi+15h] ; cope l'octet de
    l'offset 0x15
.text:64404C4F                 mov     [esi+15h], al ; dans l'offset 0x15 de
    l'objet CDwsGui
```

La fonction a été appelée depuis la fonction appelée *CDwsGui::CopyOptions* ! Encore merci pour les informations de débogage.

Mais la vraie réponse est dans *CDwsGui::Init()* :

```
.text:6440B0BF loc_6440B0BF:
.text:6440B0BF                 mov     eax, [ebp+arg_0]
.text:6440B0C2                 push    [ebp+arg_4]
.text:6440B0C5                 mov     [esi+2844h], eax
.text:6440B0CB                 lea     eax, [esi+28h] ; ESI est un pointeur
    sur l'objet CDwsGui
.text:6440B0CE                 push    eax
.text:6440B0CF                 call    CDwsGui__CopyOptions
```

Enfin, nous comprennons: le tableau rempli dans la fonction `load_command_line()` est stocké dans la classe *CDwsGui* mais à l'adresse `this+0x28`. 0x15 + 0x28 vaut exactement 0x3D. OK, nous avons trouvé le point où la valeur y est copiée.

Trouvons les autres endroits où l'offset 0x3D est utilisé. Voici l'un d'entre eux dans la fonction *CDwsGui::SapguiRun* (à nouveau, merci aux appels de débogage) :

```
.text:64409D58                 cmp     [esi+3Dh], bl   ; ESI est un pointeur
    sur l'objet CDwsGui
.text:64409D5B                 lea     ecx, [esi+2B8h]
.text:64409D61                 setz    al
.text:64409D64                 push    eax ; arg_10 de
    CConnectionContext::CreateNetwork
.text:64409D65                 push    dword ptr [esi+64h]

; nom original: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64409D68                 call    ds:mfc90_910
.text:64409D68                                   ; pas d'arguments
.text:64409D6E                 push    eax
.text:64409D6F                 lea     ecx, [esi+2BCh]

; nom orignal: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64409D75                 call    ds:mfc90_910
.text:64409D75                                   ; pas d'arguments
.text:64409D7B                 push    eax
.text:64409D7C                 push    esi
.text:64409D7D                 lea     ecx, [esi+8]
.text:64409D80                 call    CConnectionContext__CreateNetwork
```

Vérifions nos découvertes.
Remplaçons `setz al` par les instructions `xor eax, eax / nop`, effaçons la variable d'environnement TDW_NOCOMPRESS et lançons SAPGUI. Ouah! La fenêtre ennuyeuse n'est plus là (comme nous l'attendions, puisque la variable d'environnement n'est pas mise) mais dans Wireshark nous pouvons voir que les paquets réseau ne sont plus compressés! Visiblement, c'est le point où le flag de compression doit être défini dans l'objet *CConnectionContext*.

Donc, le flag de compression est passé dans le 5ème argument de *CConnectionContext::CreateNetwork*. À l'intérieur de la fonction, une autre est appelée:

```
...
.text:64403476                 push    [ebp+compression]
.text:64403479                 push    [ebp+arg_C]
.text:6440347C                 push    [ebp+arg_8]
```

```
.text:6440347F                  push    [ebp+arg_4]
.text:64403482                  push    [ebp+arg_0]
.text:64403485                  call    CNetwork__CNetwork
```

Le flag de compression est passé ici dans le 5ème argument au constructer *CNetwork::CNetwork*.

Et voici comment le constructeur *CNetwork* défini le flag dans l'objet *CNetwork* suivant son 5ème argument *et* une autre variable qui peut probablement aussi affecter la compression des paquets réseau.

```
.text:64411DF1                  cmp     [ebp+compression], esi
.text:64411DF7                  jz      short set_EAX_to_0
.text:64411DF9                  mov     al, [ebx+78h]   ; une autre valeur
    pourrait affecter la compression?
.text:64411DFC                  cmp     al, '3'
.text:64411DFE                  jz      short set_EAX_to_1
.text:64411E00                  cmp     al, '4'
.text:64411E02                  jnz     short set_EAX_to_0
.text:64411E04
.text:64411E04 set_EAX_to_1:
.text:64411E04                  xor     eax, eax
.text:64411E06                  inc     eax             ; EAX -> 1
.text:64411E07                  jmp     short loc_64411E0B
.text:64411E09
.text:64411E09 set_EAX_to_0:
.text:64411E09
.text:64411E09                  xor     eax, eax        ; EAX -> 0
.text:64411E0B
.text:64411E0B loc_64411E0B:
.text:64411E0B                  mov     [ebx+3A4h], eax ; EBX est un pointeur
    sur l'object CNetwork
```

À ce point, nous savons que le flag de compression est stocké dans la classe *CNetwork* à l'adresse *this+0x3A4*.

Plongeons-nous maintenant dans SAPguilib.dll à la recherche de la valeur `0x3A4`. Et il y a une seconde occurrence dans *CDwsGui::OnClientMessageWrite* (Merci infiniment pour les informations de débogage) :

```
.text:64406F76 loc_64406F76:
.text:64406F76                  mov     ecx, [ebp+7728h+var_7794]
.text:64406F79                  cmp     dword ptr [ecx+3A4h], 1
.text:64406F80                  jnz     compression_flag_is_zero
.text:64406F86                  mov     byte ptr [ebx+7], 1
.text:64406F8A                  mov     eax, [esi+18h]
.text:64406F8D                  mov     ecx, eax
.text:64406F8F                  test    eax, eax
.text:64406F91                  ja      short loc_64406FFF
.text:64406F93                  mov     ecx, [esi+14h]
.text:64406F96                  mov     eax, [esi+20h]
.text:64406F99
.text:64406F99 loc_64406F99:
.text:64406F99                  push    dword ptr [edi+2868h] ; int
```

```
.text:64406F9F                 lea     edx, [ebp+7728h+var_77A4]
.text:64406FA2                 push    edx             ; int
.text:64406FA3                 push    30000           ; int
.text:64406FA8                 lea     edx, [ebp+7728h+Dst]
.text:64406FAB                 push    edx             ; Dst
.text:64406FAC                 push    ecx             ; int
.text:64406FAD                 push    eax             ; Src
.text:64406FAE                 push    dword ptr [edi+28C0h] ; int
.text:64406FB4                 call    sub_644055C5        ; routine de
    compression actuelle
.text:64406FB9                 add     esp, 1Ch
.text:64406FBC                 cmp     eax, 0FFFFFFF6h
.text:64406FBF                 jz      short loc_64407004
.text:64406FC1                 cmp     eax, 1
.text:64406FC4                 jz      loc_6440708C
.text:64406FCA                 cmp     eax, 2
.text:64406FCD                 jz      short loc_64407004
.text:64406FCF                 push    eax
.text:64406FD0                 push    offset aCompressionErr ;
    "compression error [rc = %d]- program wi"...
.text:64406FD5                 push    offset aGui_err_compre ;
    "GUI_ERR_COMPRESS"
.text:64406FDA                 push    dword ptr [edi+28D0h]
.text:64406FE0                 call    SapPcTxtRead
```

Jetons un œil dans *sub_644055C5*. Nous y voyons seulement l'appel à memcpy() et une autre fonction appelée (par IDA) *sub_64417440*.

Et, regardons dans *sub_64417440*. Nous y voyons:

```
.text:6441747C                 push    offset aErrorCsrcompre ;
    "\nERROR: CsRCompress: invalid handle"
.text:64417481                 call    eax ; dword_644F94C8
.text:64417483                 add     esp, 4
```

Voilà! Nous avons trouvé la fonction qui effectue la compression des données. Comme cela a été décrit dans le passé [35],

cette fonction est utilisé dans SAP et aussi dans le projet open-source MaxDB. Donc, elle est disponible sous forme de code source.

La dernière vérification est faite ici:

```
.text:64406F79                 cmp     dword ptr [ecx+3A4h], 1
.text:64406F80                 jnz     compression_flag_is_zero
```

Remplaçons ici JNZ par un JMP inconditionnel. Supprimons la variable d'environnement TDW_NOCOMPRESS. Voilà!

Dans Wireshark nous voyons que les messages du client ne sont pas compressés. Les réponses du serveur, toutefois, le sont.

Donc nous avons trouvé le lien entre la variable d'environnement et le point où la routine de compression peut être appelée ou non.

[35] http://conus.info/utils/SAP_pkt_decompr.txt

8.12.2 Fonctions de vérification de mot de passe de SAP 6.0

Lorsque je suis retourné sur SAP 6.0 IDES installé sur une machine VMware, je me suis aperçu que j'avais oublié le mot de passe pour le compte SAP*, puis je m'en suis souvenu, mais j'ai alors eu ce message «*Password logon no longer possible - too many failed attempts*», car j'ai fait trop de tentatives avant de m'en rappeler.

La première très bonne nouvelle fût que le fichier PDB complet de *disp+work.pdb* était fourni avec SAP, et il contient presque tout: noms de fonction, structures, types, variable locale et nom d'arguments, etc. Quel cadeau somptueux!

Il y a l'utilitaire TYPEINFODUMP[36] pour convertir les fichiers PDB en quelque chose de lisible et grepable.

Voici un exemple d'information d'une fonction + ses arguments + ses variables locales:

```
FUNCTION ThVmcSysEvent
  Address:         10143190  Size:      675 bytes  Index:    60483  ↙
    ↳ TypeIndex:    60484
  Type: int NEAR_C ThVmcSysEvent (unsigned int, unsigned char, unsigned ↙
    ↳ short*)
Flags: 0
PARAMETER events
  Address: Reg335+288  Size:        4 bytes  Index:    60488  TypeIndex:    ↙
    ↳  60489
  Type: unsigned int
Flags: d0
PARAMETER opcode
  Address: Reg335+296  Size:        1 bytes  Index:    60490  TypeIndex:    ↙
    ↳  60491
  Type: unsigned char
Flags: d0
PARAMETER serverName
  Address: Reg335+304  Size:        8 bytes  Index:    60492  TypeIndex:    ↙
    ↳  60493
  Type: unsigned short*
Flags: d0
STATIC_LOCAL_VAR func
  Address:         12274af0  Size:        8 bytes  Index:    60495  ↙
    ↳ TypeIndex:    60496
  Type: wchar_t*
Flags: 80
LOCAL_VAR admhead
  Address: Reg335+304  Size:        8 bytes  Index:    60498  TypeIndex:    ↙
    ↳  60499
  Type: unsigned char*
Flags: 90
LOCAL_VAR record
  Address: Reg335+64  Size:      204 bytes  Index:    60501  TypeIndex:    ↙
    ↳ 60502
  Type: AD_RECORD
Flags: 90
```

[36] http://www.debuginfo.com/tools/typeinfodump.html

```
LOCAL_VAR adlen
  Address: Reg335+296  Size:        4 bytes  Index:    60508  TypeIndex:   ↙
    ↳ 60509
  Type: int
Flags: 90
```

Et voici un exemple d'une structure:

```
STRUCT DBSL_STMTID
Size: 120  Variables: 4  Functions: 0  Base classes: 0
MEMBER moduletype
  Type:  DBSL_MODULETYPE
  Offset:        0  Index:        3  TypeIndex:    38653
MEMBER module
  Type:  wchar_t module[40]
  Offset:        4  Index:        3  TypeIndex:      831
MEMBER stmtnum
  Type:  long
  Offset:       84  Index:        3  TypeIndex:      440
MEMBER timestamp
  Type:  wchar_t timestamp[15]
  Offset:       88  Index:        3  TypeIndex:     6612
```

Wow!

Une autre bonne nouvelle: les appels de *debugging* (il y en a beaucoup) sont très utiles.

Ici, vous pouvez remarquer la variable globale *ct_level*[37], qui reflète le niveau actuel de trace.

Il y a beaucoup d'ajout de débogage dans le fichier *disp+work.exe* :

```
cmp     cs:ct_level, 1
jl      short loc_1400375DA
call    DpLock
lea     rcx, aDpxxtool4_c ; "dpxxtool4.c"
mov     edx, 4Eh         ; line
call    CTrcSaveLocation
mov     r8, cs:func_48
mov     rcx, cs:hdl      ; hdl
lea     rdx, aSDpreadmemvalu ; "%s: DpReadMemValue (%d)"
mov     r9d, ebx
call    DpTrcErr
call    DpUnlock
```

Si le niveau courant de trace est plus élevé ou égal à la limite défini dans le code ici, un message de débogage est écrit dans les fichiers de log comme *dev_w0*, *dev_disp*, et autres fichiers *dev**.

Essayons de grepper dans le fichier que nous avons obtenu à l'aide de l'utilitaire TYPEINFODUMP:

[37] Plus d'information sur le niveau de trace: http://help.sap.com/saphelp_nwpi71/helpdata/en/46/962416a5a613e8e10000000a155369/content.htm

```
cat "disp+work.pdb.d" | grep FUNCTION | grep -i password
```

Nous obtenons:

```
FUNCTION rcui::AgiPassword::DiagISelection
FUNCTION ssf_password_encrypt
FUNCTION ssf_password_decrypt
FUNCTION password_logon_disabled
FUNCTION dySignSkipUserPassword
FUNCTION migrate_password_history
FUNCTION password_is_initial
FUNCTION rcui::AgiPassword::IsVisible
FUNCTION password_distance_ok
FUNCTION get_password_downwards_compatibility
FUNCTION dySignUnSkipUserPassword
FUNCTION rcui::AgiPassword::GetTypeName
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$2
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$0
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$1
FUNCTION usm_set_password
FUNCTION rcui::AgiPassword::TraceTo
FUNCTION days_since_last_password_change
FUNCTION rsecgrp_generate_random_password
FUNCTION rcui::AgiPassword::`scalar deleting destructor'
FUNCTION password_attempt_limit_exceeded
FUNCTION handle_incorrect_password
FUNCTION `rcui::AgiPassword::`scalar deleting destructor''::`1'::dtor$1
FUNCTION calculate_new_password_hash
FUNCTION shift_password_to_history
FUNCTION rcui::AgiPassword::GetType
FUNCTION found_password_in_history
FUNCTION `rcui::AgiPassword::`scalar deleting destructor''::`1'::dtor$0
FUNCTION rcui::AgiObj::IsaPassword
FUNCTION password_idle_check
FUNCTION SlicHwPasswordForDay
FUNCTION rcui::AgiPassword::IsaPassword
FUNCTION rcui::AgiPassword::AgiPassword
FUNCTION delete_user_password
FUNCTION usm_set_user_password
FUNCTION Password_API
FUNCTION get_password_change_for_SSO
FUNCTION password_in_USR40
FUNCTION rsec_agrp_abap_generate_random_password
```

Essayons aussi de chercher des messages de debug qui contiennent les mots *«password»* et *«locked»*. L'un d'entre eux se trouve dans la chaîne *«user was locked by subsequently failed password logon attempts»*, référencé dans la fonction *password_attempt_limit_exceeded()*.

D'autres chaînes que cette fonction peut écrire dans le fichier de log sont: *«password logon attempt will be rejected immediately (preventing dictionary attacks)»*, *«failed-logon lock: expired (but not removed due to 'read-only' operation)»*, *«failed-logon*

lock: expired => removed».

Après avoir joué un moment avec cette fonction, nous remarquons que le problème se situe exactement dedans. Elle est appelée depuis la fonction *chckpass()* —une des fonctions de vérification u mot de passe.

d'abord, nous voulons être sûrs que nous sommes au bon endroit:

Lançons tracer :

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode
```

```
PID=2236|TID=2248|(0) disp+work.exe!chckpass (0x202c770, L"Brewered1
                              ", 0x41) (called from 0x1402f1060 (disp+work.
    exe!usrexist+0x3c0))
PID=2236|TID=2248|(0) disp+work.exe!chckpass -> 0x35
```

L'enchaînement des appels est: *syssigni() -> DylSigni() -> dychkusr() -> usrexist() -> chckpass()*.

Le nombre 0x35 est une erreur renvoyée dans *chckpass()* à cet endroit:

```
.text:00000001402ED567 loc_1402ED567:                          ; CODE XREF:
    chckpass+B4
.text:00000001402ED567                 mov     rcx, rbx        ; usr02
.text:00000001402ED56A                 call    password_idle_check
.text:00000001402ED56F                 cmp     eax, 33h
.text:00000001402ED572                 jz      loc_1402EDB4E
.text:00000001402ED578                 cmp     eax, 36h
.text:00000001402ED57B                 jz      loc_1402EDB3D
.text:00000001402ED581                 xor     edx, edx        ;
    usr02_readonly
.text:00000001402ED583                 mov     rcx, rbx        ; usr02
.text:00000001402ED586                 call
    password_attempt_limit_exceeded
.text:00000001402ED58B                 test    al, al
.text:00000001402ED58D                 jz      short loc_1402ED5A0
.text:00000001402ED58F                 mov     eax, 35h
.text:00000001402ED594                 add     rsp, 60h
.text:00000001402ED598                 pop     r14
.text:00000001402ED59A                 pop     r12
.text:00000001402ED59C                 pop     rdi
.text:00000001402ED59D                 pop     rsi
.text:00000001402ED59E                 pop     rbx
.text:00000001402ED59F                 retn
```

Bien, vérifions:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!
    password_attempt_limit_exceeded,args:4,unicode,rt:0
```

```
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0
    x202c770, 0, 0x257758, 0) (called from 0x1402ed58b (disp+work.exe!
    chckpass+0xeb))
```

```
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0↙
    ↳ x202c770, 0, 0, 0) (called from 0x1402e9794 (disp+work.exe!chngpass+0↙
    ↳ xe4))
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

Excellent! Nous pouvons nous connecter avec succès maintenant.

À propos, nous pouvons prétendre que nous avons oublier le mot de passe, modifier la fonction *chckpass()* afin qu'elle renvoie toujours une valeur de 0, ce qui est suffisant pour passer outre la vérification:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode,rt↙
    ↳ :0
```

```
PID=2744|TID=360|(0) disp+work.exe!chckpass (0x202c770, L"bogus          ↙
    ↳                           ", 0x41) (called from 0x1402f1060 (disp+work.↙
    ↳ exe!usrexist+0x3c0))
PID=2744|TID=360|(0) disp+work.exe!chckpass -> 0x35
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

Ce que l'on peut aussi dire en analysant la fonction *password_attempt_limit_exceeded()*, c'est qu'à son tout début, on voit cet appel:

```
lea     rcx, aLoginFailed_us ; "login/failed_user_auto_unlock"
call    sapgparam
test    rax, rax
jz      short loc_1402E19DE
movzx   eax, word ptr [rax]
cmp     ax, 'N'
jz      short loc_1402E19D4
cmp     ax, 'n'
jz      short loc_1402E19D4
cmp     ax, '0'
jnz     short loc_1402E19DE
```

Étonnement, la fonction *sapgparam()* est utilisée pour chercher la valeur de certains paramètres de configuration. Cette fonction peut être appelée depuis 1768 endroits différents. Il semble qu'avec l'aide de cette information, nous pouvons facilement trouver les endroits dans le code, où le contrôle du flux est affecté par des configurations spécifiques de paramètres.

C'est vraiment agréable. Le nom des fonctions est très clair, bien plus que dans Oracle RDBMS. Il semble que le processus *disp+work* est écrit en C++. A-t-il été récrit il y a quelques temps?

8.13 Oracle RDBMS

8.13.1 Table V$VERSION dans Oracle RDBMS

Oracle RDBMS 11.2 est un programme gigantesque, son module principal oracle.exe contient environ 124000 fonctions. Par comparaison, le noyau de Windows 7 x86 (ntoskrnl.exe) contient environ 11000 fonctions et le noyau Linux 3.9.8 (avec les drivers par défaut compilés)—31000 fonctions.

Commençons par une question facile. Où Oracle RDBMS trouve-t-il toutes ces informations, lorsque l'on exécute une expression simple dans SQL*Plus:

```
SQL> select * from V$VERSION;
```

Et nous obtenons:

```
BANNER
--------------------------------------------------------

Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
PL/SQL Release 11.2.0.1.0 - Production
CORE    11.2.0.1.0      Production
TNS for 32-bit Windows: Version 11.2.0.1.0 - Production
NLSRTL Version 11.2.0.1.0 - Production
```

Allons-y. Où Oracle RDBMS trouve-t-il la chaîne V$VERSION ?

Dans la version win32, le fichier oracle.exe contient la chaîne, c'est facile à voir. Mais nous pouvons aussi utiliser les fichiers objet (.o) de la version Linux d'Oracle RDBMS, puisque contrairement à la version win32 oracle.exe, les noms de fonctions (et aussi les variables globales) y sont préservés.

Donc, le fichier kqf.o contient la chaîne V$VERSION. Le fichier objet se trouve dans la bibliothèque Oracle principale libserver11.a.

On trouve une référence à ce texte dans la table kqfviw stockée dans le même fichier, kqf.o :

Listing 8.10 : kqf.o

```
.rodata:0800C4A0 kqfviw dd 0Bh    ; DATA XREF: kqfchk:loc_8003A6D
.rodata:0800C4A0                  ; kqfgbn+34
.rodata:0800C4A4        dd offset _2__STRING_10102_0 ; "GV$WAITSTAT"
.rodata:0800C4A8        dd 4
.rodata:0800C4AC        dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C4B0        dd 3
.rodata:0800C4B4        dd 0
.rodata:0800C4B8        dd 195h
.rodata:0800C4BC        dd 4
.rodata:0800C4C0        dd 0
.rodata:0800C4C4        dd 0FFFFC1CBh
.rodata:0800C4C8        dd 3
.rodata:0800C4CC        dd 0
.rodata:0800C4D0        dd 0Ah
```

```
.rodata:0800C4D4         dd offset _2__STRING_10104_0 ; "V$WAITSTAT"
.rodata:0800C4D8         dd 4
.rodata:0800C4DC         dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C4E0         dd 3
.rodata:0800C4E4         dd 0
.rodata:0800C4E8         dd 4Eh
.rodata:0800C4EC         dd 3
.rodata:0800C4F0         dd 0
.rodata:0800C4F4         dd 0FFFFC003h
.rodata:0800C4F8         dd 4
.rodata:0800C4FC         dd 0
.rodata:0800C500         dd 5
.rodata:0800C504         dd offset _2__STRING_10105_0 ; "GV$BH"
.rodata:0800C508         dd 4
.rodata:0800C50C         dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C510         dd 3
.rodata:0800C514         dd 0
.rodata:0800C518         dd 269h
.rodata:0800C51C         dd 15h
.rodata:0800C520         dd 0
.rodata:0800C524         dd 0FFFFC1EDh
.rodata:0800C528         dd 8
.rodata:0800C52C         dd 0
.rodata:0800C530         dd 4
.rodata:0800C534         dd offset _2__STRING_10106_0 ; "V$BH"
.rodata:0800C538         dd 4
.rodata:0800C53C         dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C540         dd 3
.rodata:0800C544         dd 0
.rodata:0800C548         dd 0F5h
.rodata:0800C54C         dd 14h
.rodata:0800C550         dd 0
.rodata:0800C554         dd 0FFFFC1EEh
.rodata:0800C558         dd 5
.rodata:0800C55C         dd 0
```

À propos, souvent, en analysant les entrailles dOracle RDBMS, vous pouvez vous demander pourquoi les noms de fonctions et de variables globales sont si étranges.

Sans doute parce qu'Oracle RDBMS est un très vieux produit et a été développé en C dans les années 80.

Et c'était un temps où le standard C garantissait que les noms de fonction et de variable pouvaient supporter seulement jusqu'à 6 caractères incluant: «6 charactères significatifs dans un identifiant externe»[38]

Probablement que la table kqfviw contient la plupart (peut-être même toutes) des vues préfixées avec V$, qui sont des *vues fixées*, toujours présentes. Superficiellement, en remarquant la récurrence cyclique des données, nous pouvons facilement voir que chaque élément de la table kqfviw a 12 champs de 32-bit. C'est très facile de créer une structure de 12 éléments dans IDA et de l'appliquer à tous les éléments

[38] Draft ANSI C Standard (ANSI X3J11/88-090) (May 13, 1988) (yurichev.com)

de la table. Depuis Oracle RDBMS version 11.2, il y a 1023 éléments dans la table, i.e., dans celle-ci sont décrites 1023 de toutes les *vues fixées* possible.

Nous reviendrons à ce nombre plus tard.

Comme on le voit, il n'y a pas beaucoup d'information sur les nombres dans les champs. Le premier nombre est toujours égal au nom de la vue (sans le zéro de fin).

Nous savons aussi que l'information sur toutes ces vues fixes peut être récupérée depuis une *vue fixée* appelée V$FIXED_VIEW_DEFINITION (à propos, l'information pour cette vue est aussi prise dans les tables kqfviw et kqfvip.) Au fait, il y a aussi 1023 éléments dans celle-ci. Coïncidence? Non.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$VERSION';

VIEW_NAME
------------------------------
VIEW_DEFINITION
------------------------------

V$VERSION
select  BANNER from GV$VERSION where inst_id = USERENV('Instance')
```

Donc, V$VERSION est une sorte de *vue terminale* pour une autre vue appelée GV$VERSION, qui est, à son tour:

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$VERSION';

VIEW_NAME
------------------------------
VIEW_DEFINITION
------------------------------

GV$VERSION
select inst_id, banner from x$version
```

Les tables préfixées par X$ dans Oracle RDBMS sont aussi des tables de service, non documentées, qui ne peuvent pas être modifiées par l'utilisateur et qui sont rafraîchies dynamiquement.

Si nous cherchons le texte

```
select BANNER from GV$VERSION where inst_id =
USERENV('Instance')
```

... dans le fichier kqf.o, nous le trouvons dans la table kqfvip :

Listing 8.11 : kqf.o

```
.rodata:080185A0 kqfvip dd offset _2__STRING_11126_0 ; DATA XREF: kqfgvcn+18
.rodata:080185A0                                      ; kqfgvt+F
.rodata:080185A0                                      ;
    "select inst_id,decode(indx,1,'data bloc"...
.rodata:080185A4        dd offset kqfv459_c_0
```

```
.rodata:080185A8        dd 0
.rodata:080185AC        dd 0

...

.rodata:08019570        dd offset _2__STRING_11378_0 ;
    "select  BANNER from GV$VERSION where in"...
.rodata:08019574        dd offset kqfv133_c_0
.rodata:08019578        dd 0
.rodata:0801957C        dd 0
.rodata:08019580        dd offset _2__STRING_11379_0 ;
    "select inst_id,decode(bitand(cfflg,1),0"...
.rodata:08019584        dd offset kqfv403_c_0
.rodata:08019588        dd 0
.rodata:0801958C        dd 0
.rodata:08019590        dd offset _2__STRING_11380_0 ;
    "select  STATUS , NAME, IS_RECOVERY_DEST"...
.rodata:08019594        dd offset kqfv199_c_0
```

La table semble avoir 4 champs dans chaque élément. À propos, elle a 1023 éléments, encore, le nombre que nous connaissons déjà.

Le second champ pointe sur une autre table qui contient les champs de la table pour cette *vue fixée*. Comme pour V$VERSION, cette table a seulement deux éléments, le premier est 6 et le second est la chaîne BANNER (le nombre 6 est la longueur de la chaîne) et après, un élément *de fin* qui contient 0 et une chaîne C *null* :

Listing 8.12 : kqf.o

```
.rodata:080BBAC4 kqfv133_c_0 dd 6     ; DATA XREF: .rodata:08019574
.rodata:080BBAC8             dd offset _2__STRING_5017_0 ; "BANNER"
.rodata:080BBACC             dd 0
.rodata:080BBAD0             dd offset _2__STRING_0_0
```

En joignant les données des deux tables kqfviw et kqfvip, nous pouvons obtenir la déclaration SQL qui est exécutée lorsque l'utilisateur souhaite faire une requête sur une *vue fixée* spécifique.

Ainsi nous pouvons écrire un programme oracle tables[39], pour collecter toutes ces informations d'un fichier objet d'Oracle RDBMS pour Linux.

Listing 8.13 : Résultat de oracle tables

```
kqfviw_element.viewname: [V$VERSION] ?: 0x3 0x43 0x1 0xffffc085 0x4
kqfvip_element.statement: [select  BANNER from GV$VERSION where inst_id = ↙
    ↘ USERENV('Instance')]
kqfvip_element.params:
[BANNER]
```

Et:

Listing 8.14 : Résultat de oracle tables

[39] yurichev.com

```
kqfviw_element.viewname: [GV$VERSION] ?: 0x3 0x26 0x2 0xffffc192 0x1
kqfvip_element.statement: [select inst_id, banner from x$version]
kqfvip_element.params:
[INST_ID] [BANNER]
```

La *vue fixée* GV$VERSION est différente de V$VERSION seulement parce qu'elle a un champ de plus avec l'identifiant de l'*instance*.

Quoiqu'il en soit, nous allons rester avec la table X$VERSION. Tout comme les autres tables X$, elle n'est pas documentée, toutefois, nous pouvons y effectuer des requêtes:

```
SQL> select * from x$version;

ADDR           INDX    INST_ID
-------- ---------- ----------
BANNER
------------------------------

0DBAF574          0          1
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production

...
```

Cette table a des champs additionnels, comme ADDR et INDX.

En faisant défiler kqf.o dans IDA, nous pouvons repérer une autre table qui contient un pointeur sur la chaîne X$VERSION, c'est kqftab :

Listing 8.15 : kqf.o

```
.rodata:0803CAC0      dd 9                    ; element number 0x1f6
.rodata:0803CAC4      dd offset _2__STRING_13113_0 ; "X$VERSION"
.rodata:0803CAC8      dd 4
.rodata:0803CACC      dd offset _2__STRING_13114_0 ; "kqvt"
.rodata:0803CAD0      dd 4
.rodata:0803CAD4      dd 4
.rodata:0803CAD8      dd 0
.rodata:0803CADC      dd 4
.rodata:0803CAE0      dd 0Ch
.rodata:0803CAE4      dd 0FFFFC075h
.rodata:0803CAE8      dd 3
.rodata:0803CAEC      dd 0
.rodata:0803CAF0      dd 7
.rodata:0803CAF4      dd offset _2__STRING_13115_0 ; "X$KQFSZ"
.rodata:0803CAF8      dd 5
.rodata:0803CAFC      dd offset _2__STRING_13116_0 ; "kqfsz"
.rodata:0803CB00      dd 1
.rodata:0803CB04      dd 38h
.rodata:0803CB08      dd 0
.rodata:0803CB0C      dd 7
.rodata:0803CB10      dd 0
.rodata:0803CB14      dd 0FFFFC09Dh
.rodata:0803CB18      dd 2
```

```
.rodata:0803CB1C      dd 0
```

Il y a beaucoup de référence aux noms de X$-table, visiblement, à toutes les X$-tables d'Oracle RDBMS 11.2. Mais encore une fois, nous n'avons pas assez d'information.

Ce que signifie la chaîne `kqvt` n'est pas clair.

Le préfixe kq peut signifier *kernel* ou *query*.

v signifie apparemment *version* et `t`—*type* ? Difficile à dire.

Une table avec un nom similaire se trouve dans `kqf.o` :

Listing 8.16 : kqf.o

```
.rodata:0808C360 kqvt_c_0 kqftap_param <4, offset _2__STRING_19_0, 917h, 0,↙
    ↳  0, 0, 4, 0, 0>
.rodata:0808C360                                  ; DATA XREF:
    .rodata:08042680
.rodata:0808C360                                  ; "ADDR"
.rodata:0808C384          kqftap_param <4, offset _2__STRING_20_0, 0B02h, ↙
    ↳ 0, 0, 0, 4, 0, 0> ;
    "INDX"
.rodata:0808C3A8          kqftap_param <7, offset _2__STRING_21_0, 0B02h, ↙
    ↳ 0, 0, 0, 4, 0, 0> ;
    "INST_ID"
.rodata:0808C3CC          kqftap_param <6, offset _2__STRING_5017_0, 601h, ↙
    ↳ 0, 0, 0, 50h, 0, 0> ;
    "BANNER"
.rodata:0808C3F0          kqftap_param <0, offset _2__STRING_0_0, 0, 0, 0, ↙
    ↳ 0, 0, 0, 0>
```

Elle contient des informations à propos de tous les champs de la table `X$VERSION`. La seule référence à cette table est dans la table `kqftap` :

Listing 8.17 : kqf.o

```
.rodata:08042680              kqftap_element <0, offset kqvt_c_0, offset↙
    ↳  kqvrow, 0> ; element 0x1f6
```

Il est intéressant de voir que cet élément ici est `0x1f6th` (502nd), tout comme le pointeur sur la chaîne `X$VERSION` dans la table `kqftab`.

Sans doute que les tables `kqftap` et `kqftab` sont complémentaires l'une de l'autre, tout comme `kqfvip` et `kqfviw`.

Nous voyons aussi un pointeur sur la fonction `kqvrow()`. Enfin, nous obtenons quelque chose d'utile!

Nous ajoutons donc ces tables à notre utilitaire oracle tables[40]. Pour `X$VERSION` nous obtenons:

Listing 8.18 : Résultat de oracle tables

```
kqftab_element.name: [X$VERSION] ?: [kqvt] 0x4 0x4 0x4 0xc 0xffffc075 0x3
```

[40]yurichev.com

```
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[BANNER] ?: 0x601 0x0 0x0 0x0 0x50 0x0 0x0
kqftap_element.fn1=kqvrow
kqftap_element.fn2=NULL
```

Avec l'aide de tracer, il est facile de vérifier que cette fonction est appelée 6 fois par ligne (depuis la fonction `qerfxFetch()`) lorsque l'on fait une requête sur la table `X$VERSION`.

Lançons tracer en mode cc (il commente chaque instruction exécutée) :

```
tracer -a:oracle.exe bpf=oracle.exe!_kqvrow,trace:cc
```

```
_kqvrow_  proc near

var_7C    = byte ptr -7Ch
var_18    = dword ptr -18h
var_14    = dword ptr -14h
Dest      = dword ptr -10h
var_C     = dword ptr -0Ch
var_8     = dword ptr -8
var_4     = dword ptr -4
arg_8     = dword ptr  10h
arg_C     = dword ptr  14h
arg_14    = dword ptr  1Ch
arg_18    = dword ptr  20h

; FUNCTION CHUNK AT .text1:056C11A0 SIZE 00000049 BYTES

          push    ebp
          mov     ebp, esp
          sub     esp, 7Ch
          mov     eax, [ebp+arg_14] ; [EBP+1Ch]=1
          mov     ecx, TlsIndex    ; [69AEB08h]=0
          mov     edx, large fs:2Ch
          mov     edx, [edx+ecx*4] ; [EDX+ECX*4]=0xc98c938
          cmp     eax, 2           ; EAX=1
          mov     eax, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
          jz      loc_2CE1288
          mov     ecx, [eax]       ; [EAX]=0..5
          mov     [ebp+var_4], edi ; EDI=0xc98c938

loc_2CE10F6: ; CODE XREF: _kqvrow_+10A
             ; _kqvrow_+1A9
          cmp     ecx, 5           ; ECX=0..5
          ja      loc_56C11C7
          mov     edi, [ebp+arg_18] ; [EBP+20h]=0
          mov     [ebp+var_14], edx ; EDX=0xc98c938
          mov     [ebp+var_8], ebx ; EBX=0
          mov     ebx, eax         ; EAX=0xcdfe554
          mov     [ebp+var_C], esi ; ESI=0xcdfe248
```

```
loc_2CE110D: ; CODE XREF: _kqvrow_+29E00E6
        mov     edx, ds:off_628B09C[ecx*4] ; [ECX*4+628B09Ch]=0x2ce1116,
    0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, 0x2ce127a
        jmp     edx             ; EDX=0x2ce1116, 0x2ce11ac, 0x2ce11db,
    0x2ce11f6, 0x2ce1236, 0x2ce127a

loc_2CE1116: ; DATA XREF: .rdata:off_628B09C
        push    offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
        mov     ecx, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
        xor     edx, edx
        mov     esi, [ebp+var_14] ; [EBP-14h]=0xc98c938
        push    edx             ; EDX=0
        push    edx             ; EDX=0
        push    50h
        push    ecx             ; ECX=0x8a172b4
        push    dword ptr [esi+10494h] ; [ESI+10494h]=0xc98cd58
        call    _kghalf         ; tracing nested maximum level (1) reached,
    skipping this CALL
        mov     esi, ds:__imp__vsnnum ; [59771A8h]=0x61bc49e0
        mov     [ebp+Dest], eax ; EAX=0xce2ffb0
        mov     [ebx+8], eax    ; EAX=0xce2ffb0
        mov     [ebx+4], eax    ; EAX=0xce2ffb0
        mov     edi, [esi]      ; [ESI]=0xb200100
        mov     esi, ds:__imp__vsnstr ; [597D6D4h]=0x65852148, "-
    Production"
        push    esi             ; ESI=0x65852148, "- Production"
        mov     ebx, edi        ; EDI=0xb200100
        shr     ebx, 18h        ; EBX=0xb200100
        mov     ecx, edi        ; EDI=0xb200100
        shr     ecx, 14h        ; ECX=0xb200100
        and     ecx, 0Fh        ; ECX=0xb2
        mov     edx, edi        ; EDI=0xb200100
        shr     edx, 0Ch        ; EDX=0xb200100
        movzx   edx, dl         ; DL=0
        mov     eax, edi        ; EDI=0xb200100
        shr     eax, 8          ; EAX=0xb200100
        and     eax, 0Fh        ; EAX=0xb2001
        and     edi, 0FFh       ; EDI=0xb200100
        push    edi             ; EDI=0
        mov     edi, [ebp+arg_18] ; [EBP+20h]=0
        push    eax             ; EAX=1
        mov     eax, ds:__imp__vsnban ;
    [597D6D8h]=0x65852100, "Oracle Database 11g Enterprise Edition Release %d.%d.%d.%d.%d %s"
        push    edx             ; EDX=0
        push    ecx             ; ECX=2
        push    ebx             ; EBX=0xb
        mov     ebx, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
        push    eax             ;
    EAX=0x65852100, "Oracle Database 11g Enterprise Edition Release %d.%d.%d.%d.%d %s"
        mov     eax, [ebp+Dest] ; [EBP-10h]=0xce2ffb0
        push    eax             ; EAX=0xce2ffb0
        call    ds:__imp__sprintf ; op1=MSVCR80.dll!sprintf tracing nested
    maximum level (1) reached, skipping this CALL
        add     esp, 38h
```

```
        mov     dword ptr [ebx], 1

loc_2CE1192: ; CODE XREF: _kqvrow_+FB
             ; _kqvrow_+128 ...
        test    edi, edi        ; EDI=0
        jnz     __VInfreq__kqvrow
        mov     esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
        mov     edi, [ebp+var_4] ; [EBP-4]=0xc98c938
        mov     eax, ebx        ; EBX=0xcdfe554
        mov     ebx, [ebp+var_8] ; [EBP-8]=0
        lea     eax, [eax+4]    ; [EAX+4]=0xce2ffb0, "NLSRTL Version
    11.2.0.1.0 - Production", "Oracle Database 11g Enterprise Edition Release
    11.2.0.1.0 - Production", "PL/SQL Release 11.2.0.1.0 - Production", "TNS
    for 32-bit Windows: Version 11.2.0.1.0 - Production"

loc_2CE11A8: ; CODE XREF: _kqvrow_+29E00F6
        mov     esp, ebp
        pop     ebp
        retn                    ; EAX=0xcdfe558

loc_2CE11AC: ; DATA XREF: .rdata:0628B0A0
        mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0, "Oracle Database 11g
    Enterprise Edition Release 11.2.0.1.0 - Production"
        mov     dword ptr [ebx], 2
        mov     [ebx+4], edx    ; EDX=0xce2ffb0, "Oracle Database 11g
    Enterprise Edition Release 11.2.0.1.0 - Production"
        push    edx             ; EDX=0xce2ffb0, "Oracle Database 11g
    Enterprise Edition Release 11.2.0.1.0 - Production"
        call    _kkxvsn         ; tracing nested maximum level (1) reached,
    skipping this CALL
        pop     ecx
        mov     edx, [ebx+4]    ; [EBX+4]=0xce2ffb0, "PL/SQL Release
    11.2.0.1.0 - Production"
        movzx   ecx, byte ptr [edx] ; [EDX]=0x50
        test    ecx, ecx        ; ECX=0x50
        jnz     short loc_2CE1192
        mov     edx, [ebp+var_14]
        mov     esi, [ebp+var_C]
        mov     eax, ebx
        mov     ebx, [ebp+var_8]
        mov     ecx, [eax]
        jmp     loc_2CE10F6

loc_2CE11DB: ; DATA XREF: .rdata:0628B0A4
        push    0
        push    50h
        mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0, "PL/SQL Release
    11.2.0.1.0 - Production"
        mov     [ebx+4], edx    ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0
    - Production"
        push    edx             ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0
    - Production"
        call    _lmxver         ; tracing nested maximum level (1) reached,
    skipping this CALL
        add     esp, 0Ch
        mov     dword ptr [ebx], 3
        jmp     short loc_2CE1192
```

```
loc_2CE11F6: ; DATA XREF: .rdata:0628B0A8
         mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0
         mov     [ebp+var_18], 50h
         mov     [ebx+4], edx    ; EDX=0xce2ffb0
         push    0
         call    _npinli         ; tracing nested maximum level (1) reached,
    skipping this CALL
         pop     ecx
         test    eax, eax        ; EAX=0
         jnz     loc_56C11DA
         mov     ecx, [ebp+var_14] ; [EBP-14h]=0xc98c938
         lea     edx, [ebp+var_18] ; [EBP-18h]=0x50
         push    edx             ; EDX=0xd76c93c
         push    dword ptr [ebx+8] ; [EBX+8]=0xce2ffb0
         push    dword ptr [ecx+13278h] ; [ECX+13278h]=0xacce190
         call    _nrtnsvrs       ; tracing nested maximum level (1) reached,
    skipping this CALL
         add     esp, 0Ch

loc_2CE122B: ; CODE XREF: _kqvrow_+29E0118
         mov     dword ptr [ebx], 4
         jmp     loc_2CE1192

loc_2CE1236: ; DATA XREF: .rdata:0628B0AC
         lea     edx, [ebp+var_7C] ; [EBP-7Ch]=1
         push    edx             ; EDX=0xd76c8d8
         push    0
         mov     esi, [ebx+8]    ; [EBX+8]=0xce2ffb0, "TNS for 32-bit
    Windows: Version 11.2.0.1.0 - Production"
         mov     [ebx+4], esi    ; ESI=0xce2ffb0, "TNS for 32-bit Windows:
    Version 11.2.0.1.0 - Production"
         mov     ecx, 50h
         mov     [ebp+var_18], ecx ; ECX=0x50
         push    ecx             ; ECX=0x50
         push    esi             ; ESI=0xce2ffb0, "TNS for 32-bit Windows:
    Version 11.2.0.1.0 - Production"
         call    _lxvers         ; tracing nested maximum level (1) reached,
    skipping this CALL
         add     esp, 10h
         mov     edx, [ebp+var_18] ; [EBP-18h]=0x50
         mov     dword ptr [ebx], 5
         test    edx, edx        ; EDX=0x50
         jnz     loc_2CE1192
         mov     edx, [ebp+var_14]
         mov     esi, [ebp+var_C]
         mov     eax, ebx
         mov     ebx, [ebp+var_8]
         mov     ecx, 5
         jmp     loc_2CE10F6

loc_2CE127A: ; DATA XREF: .rdata:0628B0B0
         mov     edx, [ebp+var_14] ; [EBP-14h]=0xc98c938
         mov     esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
         mov     edi, [ebp+var_4] ; [EBP-4]=0xc98c938
```

```
        mov     eax, ebx        ; EBX=0xcdfe554
        mov     ebx, [ebp+var_8] ; [EBP-8]=0

loc_2CE1288: ; CODE XREF: _kqvrow_+1F
        mov     eax, [eax+8]    ; [EAX+8]=0xce2ffb0, "NLSRTL Version
    11.2.0.1.0 - Production"
        test    eax, eax        ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0
    - Production"
        jz      short loc_2CE12A7
        push    offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
        push    eax             ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0
    - Production"
        mov     eax, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
        push    eax             ; EAX=0x8a172b4
        push    dword ptr [edx+10494h] ; [EDX+10494h]=0xc98cd58
        call    _kghfrf         ; tracing nested maximum level (1) reached,
    skipping this CALL
        add     esp, 10h

loc_2CE12A7: ; CODE XREF: _kqvrow_+1C1
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        retn                    ; EAX=0
_kqvrow_  endp
```

Maintenant, il est facile de voir que le nombre est passé de l'extérieur. La fonction renvoie une chaîne, construite comme ceci:

String 1	Using vsnstr, vsnnum, vsnban global variables. Calls sprintf().
String 2	Calls kkxvsn().
String 3	Calls lmxver().
String 4	Calls npinli(), nrtnsvrs().
String 5	Calls lxvers().

C'est ainsi que les fonctions correspondantes sont appelées pour déterminer la version de chaque module.

8.13.2 Table X$KSMLRU dans Oracle RDBMS

Il y a une mention d'une table spéciale dans la note *Diagnosing and Resolving Error ORA-04031 on the Shared Pool or Other Memory Pools [Video] [ID 146599.1]* :

> Il y a une table fixée appelée X$KSMLRU qui suit les différentes allocations dans le pool partagé qui force les autres objets du pool partagé à vieillir. Cette table fixée peut être utilisée pour identifier ce qui cause une grosse allocation.
>
> Si plusieurs objets sont supprimés périodiquement du pool partagé, alors ceci va poser des problèmes de temps de réponse et va probablement provoquer des problèmes de contention du verrou de cache de bibliothèque lorsque les objets seront rechargés dans le pool partagé.

> Une chose inhabituelle à propos de la table fixée X$KSMLRU est que le contenu de la table fixée est écrasé à chaque fois que quelqu'uni effectue requête dans la table fixée. Ceci est fait puisque la table fixée ne contient que l'allocation la plus large qui s'est produite. Les valeurs sont réinitialisées après avoir été sélectionnées, de sorte que les allocations importantes suivantes puissent ête inscrites, même si elles ne sont pas aussi laarges que celles qui se sont produites précédemment. À cause de cette réinitialisation, la sortie produite par la sélection de cette table doit être soigneusement conservée puisqu'elle ne peut plus être récupérée après que la requête a été faite.

Toutefois, comme on peut le vérifier facilement, le contenu de cette table est effacé à chaque fois qu'on l'interroge. Pouvons-nous trouver pourquoi? Retournons aux tables que nous connaissons déjà: `kqftab` et `kqftap` qui sont générées avec l'aide d'oracle tables[41], qui a toutes les informations concernant les table X$-. Nous pouvons voir ici que la fonction `ksmlrs()` est appelée pour préparer les éléments de cette table:

Listing 8.19 : Résultat de oracle tables

```
kqftab_element.name: [X$KSMLRU] ?: [ksmlr] 0x4 0x64 0x11 0xc 0xffffc0bb 0x5
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRIDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRDUR] ?: 0xb02 0x0 0x0 0x0 0x4 0x4 0x0
kqftap_param.name=[KSMLRSHRPOOL] ?: 0xb02 0x0 0x0 0x0 0x4 0x8 0x0
kqftap_param.name=[KSMLRCOM] ?: 0x501 0x0 0x0 0x0 0x14 0xc 0x0
kqftap_param.name=[KSMLRSIZ] ?: 0x2 0x0 0x0 0x0 0x4 0x20 0x0
kqftap_param.name=[KSMLRNUM] ?: 0x2 0x0 0x0 0x0 0x4 0x24 0x0
kqftap_param.name=[KSMLRHON] ?: 0x501 0x0 0x0 0x0 0x20 0x28 0x0
kqftap_param.name=[KSMLROHV] ?: 0xb02 0x0 0x0 0x0 0x4 0x48 0x0
kqftap_param.name=[KSMLRSES] ?: 0x17 0x0 0x0 0x0 0x4 0x4c 0x0
kqftap_param.name=[KSMLRADU] ?: 0x2 0x0 0x0 0x0 0x4 0x50 0x0
kqftap_param.name=[KSMLRNID] ?: 0x2 0x0 0x0 0x0 0x4 0x54 0x0
kqftap_param.name=[KSMLRNSD] ?: 0x2 0x0 0x0 0x0 0x4 0x58 0x0
kqftap_param.name=[KSMLRNCD] ?: 0x2 0x0 0x0 0x0 0x4 0x5c 0x0
kqftap_param.name=[KSMLRNED] ?: 0x2 0x0 0x0 0x0 0x4 0x60 0x0
kqftap_element.fn1=ksmlrs
kqftap_element.fn2=NULL
```

En effet, avec l'aide de tracer, il est facile de voir que cette fonction est appelée à chaque fois que nous interrogeons la table X$KSMLRU.

Ici nous voyons une référence aux fonctions `ksmsplu_sp()` et `ksmsplu_jp()`, chacune d'elles appelle `ksmsplu()` à la fin. À la fin de la fonction `ksmsplu()` nous voyons un appel à `memset()` :

Listing 8.20 : ksm.o

```
...
```

[41] yurichev.com

```
.text:00434C50 loc_434C50:    ; DATA XREF: .rdata:off_5E50EA8
.text:00434C50         mov     edx, [ebp-4]
.text:00434C53         mov     [eax], esi
.text:00434C55         mov     esi, [edi]
.text:00434C57         mov     [eax+4], esi
.text:00434C5A         mov     [edi], eax
.text:00434C5C         add     edx, 1
.text:00434C5F         mov     [ebp-4], edx
.text:00434C62         jnz     loc_434B7D
.text:00434C68         mov     ecx, [ebp+14h]
.text:00434C6B         mov     ebx, [ebp-10h]
.text:00434C6E         mov     esi, [ebp-0Ch]
.text:00434C71         mov     edi, [ebp-8]
.text:00434C74         lea     eax, [ecx+8Ch]
.text:00434C7A         push    370h            ; Size
.text:00434C7F         push    0               ; Val
.text:00434C81         push    eax             ; Dst
.text:00434C82         call    __intel_fast_memset
.text:00434C87         add     esp, 0Ch
.text:00434C8A         mov     esp, ebp
.text:00434C8C         pop     ebp
.text:00434C8D         retn
.text:00434C8D _ksmsplu  endp
```

Des constructions comme `memset (block, 0, size)` sont souvent utilisées pour mettre à zéro un bloc de mémoire. Que se passe-t-il si nous prenons le risque de bloquer l'appel à `memset (block, 0, size)` et regardons ce qui se produit?

Lançons tracer avec les options suivantes: mettre un point d'arrêt en 0x434C7A (le point où les arguments sont passés à `memset()`), afin que tracer mette le compteur de programme EIP au point où les arguments passés à `memset()` sont éffacés (en 0x434C8A). On peut dire que nous simulons juste un saut inconditionnel de l'adresse 0x434C7A à 0x434C8A.

```
tracer -a:oracle.exe bpx=oracle.exe!0x00434C7A,set(eip,0x00434C8A)
```

(Important: toutes ces adresses sont valides seulement pour la version win32 de Oracle RDBMS 11.2)

En effet, nous pouvons maintenant interroger la table X$KSMLRU autant de fois que nous voulons et elle n'est plus du tout effacée!

Au cas où, n'essayez pas ceci sur vos serveurs de production.

Ce n'est probablement pas un comportement très utile ou souhaité, mais comme une expérience pour déterminer l'emplacement d'un bout de code dont nous avons besoin, ça remplit parfaitement notre besoin!

8.13.3 Table V$TIMER dans Oracle RDBMS

V$TIMER est une autre *vue fixée* qui reflète une valeur changeant rapidement:

> V$TIMER affiche le temps écoulé en centièmes de seconde. Le temps est mesuré depuis le début de the epoch, qui est dépendant du système d'exploitation, et qui redevient 0 si la valeur déborde quatre octets (environ 497 jours).

(From Oracle RDBMS documentation [42])

Il est intéressant que les périodes soient différentes pour Oracle pour win32 et pour Linux. Allons-nous réussir à trouver la fonction qui génère cette valeur?

On voit que cette valeur est finalement prise de la table X$KSUTM.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$TIMER';

VIEW_NAME
------------------------------
VIEW_DEFINITION
------------------------------

V$TIMER
select  HSECS from GV$TIMER where inst_id = USERENV('Instance')

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$TIMER';

VIEW_NAME
------------------------------
VIEW_DEFINITION
------------------------------

GV$TIMER
select inst_id,ksutmtim from x$ksutm
```

Nous maintenant bloqué par un petit problème, il n'y a pas de référence à une ou des fonction(s) générants des valeurs dans les tables kqftab/kqftap :

Listing 8.21 : Résultat de oracle tables

```
kqftab_element.name: [X$KSUTM] ?: [ksutm] 0x1 0x4 0x4 0x0 0xffffc09b 0x3
kqftap_param.name=[ADDR] ?: 0x10917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0x20b02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSUTMTIM] ?: 0x1302 0x0 0x0 0x0 0x4 0x0 0x1e
kqftap_element.fn1=NULL
kqftap_element.fn2=NULL
```

Lorsque nous essayons de trouver la chaîne KSUTMTIM, nous la voyons dans cette fonction:

```
kqfd_DRN_ksutm_c proc near    ; DATA XREF: .rodata:0805B4E8
```

[42] http://docs.oracle.com/cd/B28359_01/server.111/b28320/dynviews_3104.htm

```
arg_0   = dword ptr  8
arg_8   = dword ptr  10h
arg_C   = dword ptr  14h

        push    ebp
        mov     ebp, esp
        push    [ebp+arg_C]
        push    offset ksugtm
        push    offset _2__STRING_1263_0 ; "KSUTMTIM"
        push    [ebp+arg_8]
        push    [ebp+arg_0]
        call    kqfd_cfui_drain
        add     esp, 14h
        mov     esp, ebp
        pop     ebp
        retn
kqfd_DRN_ksutm_c endp
```

La fonction kqfd_DRN_ksutm_c() est mentionnée dans la table kqfd_tab_registry_0 :

```
dd offset _2__STRING_62_0 ; "X$KSUTM"
dd offset kqfd_OPN_ksutm_c
dd offset kqfd_tabl_fetch
dd 0
dd 0
dd offset kqfd_DRN_ksutm_c
```

Il y a une fonction ksugtm() référencée ici. Voyons ce qu'elle contient dans (Linux x86) :

Listing 8.22 : ksu.o

```
ksugtm  proc near

var_1C  = byte ptr -1Ch
arg_4   = dword ptr  0Ch

        push    ebp
        mov     ebp, esp
        sub     esp, 1Ch
        lea     eax, [ebp+var_1C]
        push    eax
        call    slgcs
        pop     ecx
        mov     edx, [ebp+arg_4]
        mov     [edx], eax
        mov     eax, 4
        mov     esp, ebp
        pop     ebp
        retn
ksugtm  endp
```

Le code dans la version win32 est presque le même.

Est-ce la fonction que nous cherchons? Regardons:

```
tracer -a:oracle.exe bpf=oracle.exe!_ksugtm,args:2,dump_args:0x4
```

Essayons encore:

```
SQL> select * from V$TIMER;

     HSECS
----------
  27294929

SQL> select * from V$TIMER;

     HSECS
----------
  27295006

SQL> select * from V$TIMER;

     HSECS
----------
  27295167
```

Listing 8.23 : Sortie de tracer

```
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!⤦
    ↳ __VInfreq__qerfxFetch+0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                           "8.               ⤦
    ↳ "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: D1 7C A0 01                                     ".|..             ⤦
    ↳ "
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!⤦
    ↳ __VInfreq__qerfxFetch+0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                           "8.               ⤦
    ↳ "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: 1E 7D A0 01                                     ".}..             ⤦
    ↳ "
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!⤦
    ↳ __VInfreq__qerfxFetch+0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                           "8.               ⤦
    ↳ "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: BF 7D A0 01                                     ".}..             ⤦
    ↳ "
```

En effet—la valeur est la même que celle que nous voyons dans SQL*Plus et elle est renvoyée dans le second argument.

Regardons ce que `slgcs()` contient (Linux x86) :

```
slgcs   proc near

var_4   = dword ptr -4
arg_0   = dword ptr  8

        push    ebp
        mov     ebp, esp
        push    esi
        mov     [ebp+var_4], ebx
        mov     eax, [ebp+arg_0]
        call    $+5
        pop     ebx
        nop                     ; PIC mode
        mov     ebx, offset _GLOBAL_OFFSET_TABLE_
        mov     dword ptr [eax], 0
        call    sltrgatime64    ; PIC mode
        push    0
        push    0Ah
        push    edx
        push    eax
        call    __udivdi3       ; PIC mode
        mov     ebx, [ebp+var_4]
        add     esp, 10h
        mov     esp, ebp
        pop     ebp
        retn
slgcs   endp
```

(c'est simplement un appel à `sltrgatime64()`

et la division de son résultat par 10 (3.12 on page 636))

Et la version win32:

```
_slgcs  proc near       ; CODE XREF: _dbgefgHtElResetCount+15
                        ; _dbgerRunActions+1528
        db      66h
        nop
        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+8]
        mov     dword ptr [eax], 0
        call    ds:__imp__GetTickCount@0 ; GetTickCount()
        mov     edx, eax
        mov     eax, 0CCCCCCCDh
        mul     edx
        shr     edx, 3
        mov     eax, edx
        mov     esp, ebp
        pop     ebp
```

```
        retn
_slgcs  endp
```

Il s'agit simplement du résultat de `GetTickCount()` [43] divisé par 10 (3.12 on page 636).

Voilà! C'est pourquoi la version win32 et Linux x86 montrent des résultats différents, car ils sont générés par des fonctions de l'OS différentes.

Drain implique apparemment de *connecter* une colonne de table spécifique à une fonction spécifique.

Nous allons ajouter le support de la table `kqfd_tab_registry_0` à oracle tables[44], maintenant nous voyons comment les variables des colonnes de table sont *connectée* à une fonction spécifique:

```
[X$KSUTM] [kqfd_OPN_ksutm_c] [kqfd_tabl_fetch] [NULL] [NULL] [⤦
    ↳ kqfd_DRN_ksutm_c]
[X$KSUSGIF] [kqfd_OPN_ksusg_c] [kqfd_tabl_fetch] [NULL] [NULL] [⤦
    ↳ kqfd_DRN_ksusg_c]
```

OPN, signifie apparemment *open*, et *DRN*, apparemment *drain*.

8.14 Code assembleur écrit à la main

8.14.1 Fichier test EICAR

Ce fichier .COM est destiné à tester les logiciels anti-virus, il est possible de le lancer sous MS-DOS et il affiche cette chaîne: «EICAR-STANDARD-ANTIVIRUS-TEST-FILE!» [45].

Une de ses propriété importante est qu'il est entièrement composé de symboles ASCII affichables, qui, de fait, permet de le créer dans n'importe quel éditeur de texte:

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

Décompilons-le:

```
; conditions initiales: SP=0FFFEh, SS:[SP]=0
0100 58            pop     ax
; AX=0, SP=0
0101 35 4F 21      xor     ax, 214Fh
; AX = 214Fh et SP = 0
0104 50            push    ax
; AX = 214Fh, SP = FFFEh et SS:[FFFE] = 214Fh
0105 25 40 41      and     ax, 4140h
; AX = 140h, SP = FFFEh et SS:[FFFE] = 214Fh
0108 50            push    ax
; AX = 140h, SP = FFFCh, SS:[FFFC] = 140h et SS:[FFFE] = 214Fh
```

[43] MSDN
[44] yurichev.com
[45] Wikipédia

```
0109 5B             pop     bx
; AX = 140h, BX = 140h, SP = FFFEh et SS:[FFFE] = 214Fh
010A 34 5C          xor     al, 5Ch
; AX = 11Ch, BX = 140h, SP = FFFEh et SS:[FFFE] = 214Fh
010C 50             push    ax
010D 5A             pop     dx
; AX = 11Ch, BX = 140h, DX = 11Ch, SP = FFFEh et SS:[FFFE] = 214Fh
010E 58             pop     ax
; AX = 214Fh, BX = 140h, DX = 11Ch et SP = 0
010F 35 34 28       xor     ax, 2834h
; AX = 97Bh, BX = 140h, DX = 11Ch et SP = 0
0112 50             push    ax
0113 5E             pop     si
; AX = 97Bh, BX = 140h, DX = 11Ch, SI = 97Bh et SP = 0
0114 29 37          sub     [bx], si
0116 43             inc     bx
0117 43             inc     bx
0118 29 37          sub     [bx], si
011A 7D 24          jge     short near ptr word_10140
011C 45 49 43 ...   db 'EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$'
0140 48 2B   word_10140  dw 2B48h  ; CD 21 (INT 21) sera ici
0142 48 2A               dw 2A48h  ; CD 20 (INT 20) sera ici
0144 0D                  db  0Dh
0145 0A                  db  0Ah
```

J'ai ajouté des commentaires à propos des registres et de la pile après chaque instruction.

En gros, toutes ces instructions sont là seulement pour exécuter ce code:

```
B4 09     MOV AH, 9
BA 1C 01  MOV DX, 11Ch
CD 21     INT 21h
CD 20     INT 20h
```

INT 21h avec la 9ème fonction (passée dans AH) affiche simplement une chaîne, dont l'adresse est passée dans DS:DX. À propos, la chaîne doit être terminée par le signe '$'. Apparemment, c'est hérité de CP/M et cette fonction a été laissée dans DOS pour la compatibilité. INT 20h renvoie au DOS.

Mais comme on peut le voir, l'opcode de cette instruction n'est pas strictement affichable. Donc, la partie principale du fichier EICAR est:

- prépare les valeurs du registre dont nous avons besoin (AH et DX) ;
- prépare les opcodes INT 21 et INT 20 en mémoire;
- exécute INT 21 et INT 20.

À propos, cette technique est largement utilisée dans la construction de shellcode, lorsque l'on doit passer le code x86 sous la forme d'une chaîne.

Voici aussi une liste de toutes les instructions x86 qui ont des opcodes affichables: .1.6 on page 1345.

8.15 Démos

Les démos (ou démonstrations?) étaient un excellent moyen de s'éxercer en mathématiques, programmation graphique et code x86 pointu.

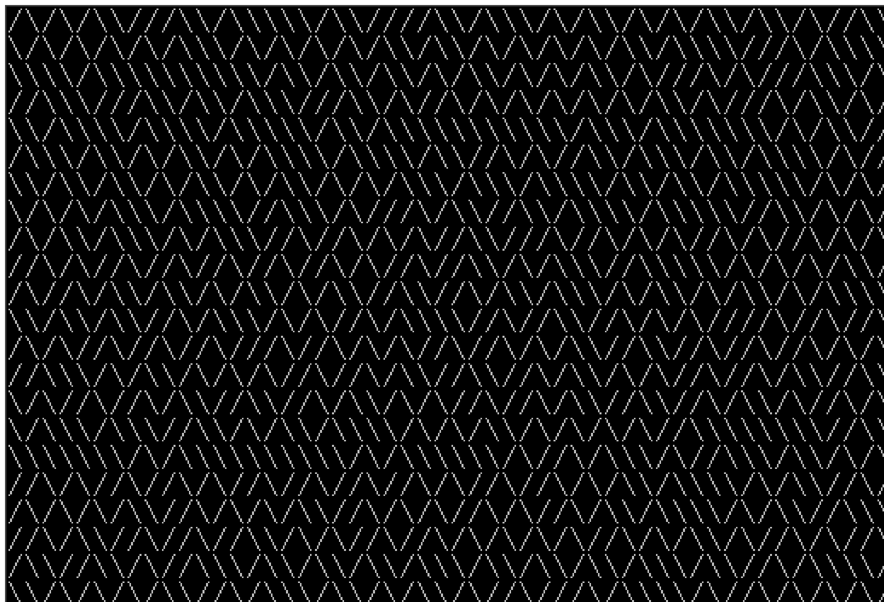
8.15.1 10 PRINT CHR$(205.5+RND(1)); : GOTO 10

Tous les exemples sont des fichier MS-DOS .COM.

Dans [Nick Montfort et al, *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*, (The MIT Press:2012)] [46]

nous pouvons nous renseigner sur l'un des générateurs de labyrinthe le plus simple possible.

Il affiche simplement un caractère slash ou backslash aléatoirement et indéfiniment, donnant quelque chose comme ceci:



Il y a quelques implémentations connues en x86 16-bit.

Version de Trixter en 42 octets

Le listing provient du site web[47], mais les commentaires sont miens.

```
00000000: B001          mov         al,1         ; mettre le mode vidéo 40x25
00000002: CD10          int         010
00000004: 30FF          xor         bh,bh        ; mettre la page vidéo pour
    l'appel int 10h
```

[46] Aussi disponible en http://trope-tank.mit.edu/10_PRINT_121114.pdf
[47] http://trixter.oldskool.org/2012/12/17/maze-generation-in-thirteen-bytes/

```
00000006: B9D007        mov         cx,007D0    ; 2000 caractères sur la
    sortie
00000009: 31C0          xor         ax,ax
0000000B: 9C            pushf                   ; pousser les flags
; prendre une valeur aléatoire du chip timer
0000000C: FA            cli                     ; interdire les interruptions
0000000D: E643          out         043,al      ; écrire 0 sur le port 43h
; lire une valeur 16-bit depuis le port 40h
0000000F: E440          in          al,040
00000011: 88C4          mov         ah,al
00000013: E440          in          al,040
00000015: 9D            popf                    ; autoriser les interruptions
    en restaurant le flag IF
00000016: 86C4          xchg        ah,al
; ici nous avons une valeur 16-bit pseudo-aléatoire
00000018: D1E8          shr         ax,1
0000001A: D1E8          shr         ax,1
; CF contient le second bit de la valeur
0000001C: B05C          mov         al,05C ;'´
; si CF=1, sauter l'instruction suivante
0000001E: 7202          jc          000000022
; si CF=0, recharger le registre AL avec un autre caractère
00000020: B02F          mov         al,02F ;'/'
; caractère de sortie
00000022: B40E          mov         ah,00E
00000024: CD10          int         010
00000026: E2E1          loop        000000009 ; boucler 2000 fois
00000028: CD20          int         020       ; sortir dans le DOS
```

La valeur pseudo aléatoire ici est en fait le temps qui a passé depuis le démarrage du système, pris depuis le temps du chip 8253, dont la valeur est incrémentée 18,2 fois par seconde.

En écrivant zéro sur le port 43h, nous envoyons la commande «select counter 0», ”counter latch”, ”binary counter” (pas une valeur BCD).

Les interruptions sont ré-autorisées avec l'instruction POPF, qui restaure aussi le flag IF.

Il n'est pas possible d'utiliser l'instruction IN avec des registres autres que AL, d'où le mélange.

Ma tentative de réduire la version de Trixter: 27 octets

Nous pouvons dire que puisque nous utilisons le timer non pas pour avoir une valeur précise, mais une valeur pseudo aléatoire, nous n'avons pas besoin de passer du temps (et du code) pour interdire les interruptions.

Une autre chose que l'on peut dire est que nous n'avons besoin que d'un bit de la partie basse 8-bit, donc lisons-le seulement.

Nous pouvons réduire légèrement le code et obtenons 27 octets:

```
00000000: B9D007    mov    cx,007D0 ; limiter la sortie à 2000 caractères
00000003: 31C0      xor    ax,ax    ; commande pour le chip timer
```

```
00000005: E643     out    043,al
00000007: E440     in     al,040   ; lire 8-bit du timer
00000009: D1E8     shr    ax,1     ; mettre le second bit dans le flag CF
0000000B: D1E8     shr    ax,1
0000000D: B05C     mov    al,05C   ; préparer '\'
0000000F: 7202     jc     000000013
00000011: B02F     mov    al,02F   ; préparer '/'
; output character to screen
00000013: B40E     mov    ah,00E
00000015: CD10     int    010
00000017: E2EA     loop   000000003
; exit to DOS
00000019: CD20     int    020
```

Prendre le contenu résiduel de la mémoire comme source d'aléas

Puisqu'il s'agit de MS-DOS, il n'y a pas du tout de protection de la mémoire, nous pouvons lire l'adresse que nous voulons. Encore mieux que ça: une seule instruction LODSB lit un octet depuis l'adresse DS:SI, mais ce n'est pas un problème si les valeurs des registres ne sont pas définies, lisons 1) des octets aléatoires; 2) depuis un endroit aléatoire de la mémoire!

Il est suggéré dans la page web de Trixter[48]d'utiliser LODSB sans aucune initialisation.

Il est aussi suggéré que l'on utilisé à la place l'instruction SCASB, car elle met les flags suivant l'octet qu'elle lit.

Une autre idée pour minimiser le code est d'utiliser l'appel système DOS INT 29h, qui affiche simplement le caractère stocké dans le registre AL.

C'est ce que Peter Ferrie a fait [49] :

Listing 8.24 : Peter Ferrie: 10 octets

```
; AL est aléatoire à ce point
00000000: AE          scasb
; CF est mis suivant le résultat de la soustraction
; de l'octet aléatoire de la mémoire de AL.
; donc c'est quelque peu aléatoire ici
00000001: D6          setalc
; AL est mis à 0xFF si CF=1 ou à 0 autrement
00000002: 242D        and         al,02D ;'-'
; AL vaut ici 0x2D ou 0
00000004: 042F        add         al,02F ;'/'
; AL vaut ici 0x5C ou 0x2F
00000006: CD29        int         029       ; afficher AL sur l'écran
00000008: EBF6        jmps        000000000 ; boucler indéfiniment
```

Donc il est possible de se passer complètement de saut conditionnel. Le code ASCII du backslash («\») est 0x5C et 0x2F pour le slash («/»). Dons nous devons convertir un bit (pseudo aléatoire) dans le flag CF en une valeur 0x5C ou 0x2F.

[48]http://trixter.oldskool.org/2012/12/17/maze-generation-in-thirteen-bytes/
[49]http://pferrie.host22.com/misc/10print.htm

Ceci est fait facilement: en AND-ant tous les bits de AL (où tous les 8 bits sont mis ou effacés) nous obtenons 0 ou 0x2D.

En ajoutant 0x2F à cette valeur, nous obtenons 0x5C ou 0x2F.

Puis il suffit de l'afficher sur l'écran.

Conclusion

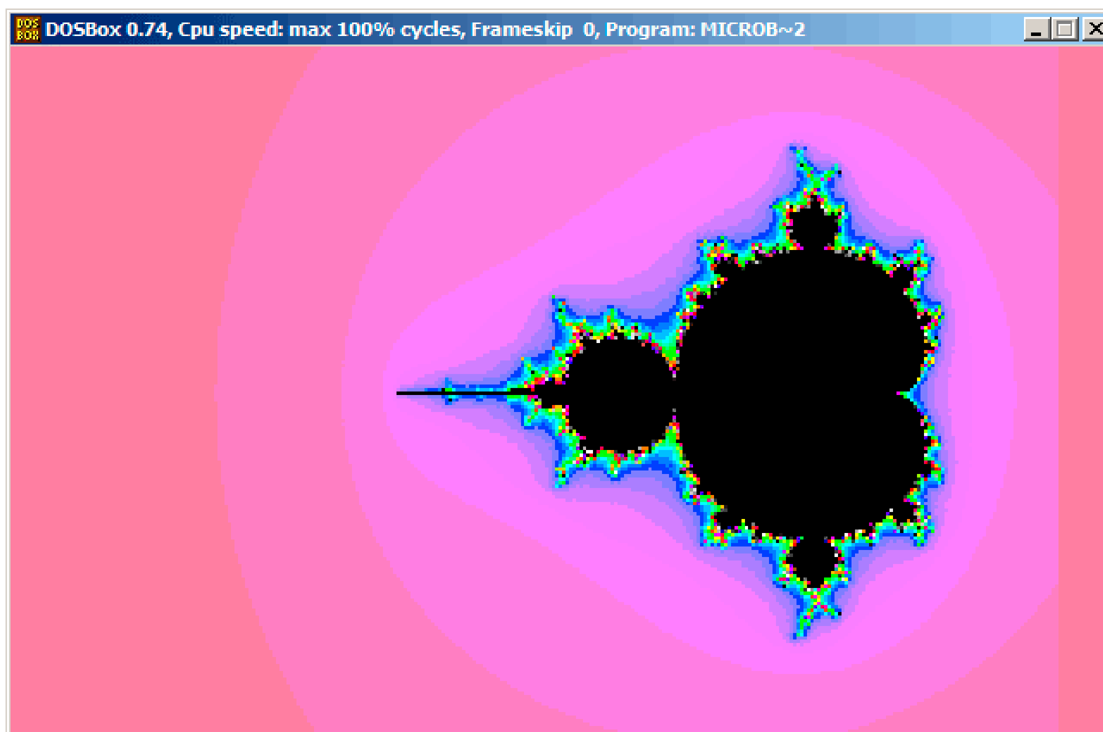
Il vaut la peine de mentionner que le résultat peut être différent dans DOSBox, Windows NT et même MS-DOS,

à cause de conditions différentes: le chip timer peut être émulé différemment et le contenu initial du registre peut être différent aussi.

8.15.2 Ensemble de Mandelbrot

Voici une démo[50] écrite par «Sir_Lagsalot» en 2009, qui dessine l'ensemble de Mandelbrot, qui est juste un programme x86, dont l'exécutable a une taille de seulement 64 octets. Il y a seulement 30 instructions x86 16-bit.

Voici ce qu'elle génère:



Essayons de comprendre comment ça fonctionne.

La démo, bien que minuscule (seulement 64 octets ou 30 instructions), implémente l'algorithme standard décrit ici, mais utilise des astuces de programmation.

Le code source est facile à télé-charger, donc le voici, mais j'ai ajouté des commentaires:

Listing 8.25 : Code source commenté

```
; X est une colonne de l'écran
; Y est une ligne de l'écran


; X=0, Y=0 X=319, Y=0
; +------------------------------->
; |
; |
```

[50]Téléchargeable ici,

```
; |
; |
; |
; |
; v
; X=0, Y=199 X=319, Y=199


; mettre le mode graphique VGA 320*200*256
mov al,13h
int 10h
; initialiser BX à 0
; initialiser DI à 0xFFFE
; DS:BX (ou DS:0) pointe à ce moment sur le Program Segment Prefix
; ... dont les 4 premiers octets sont CD 20 FF 9F
les ax,[bx]
; ES:AX=9FFF:20CD

FillLoop:
; mettre DX à 0. CWD fonctionne comme: DX:AX = sign_extend(AX).
; AX ici contient 0x20CD (au début) ou moins que 320 (lorsque l'on revient lors de la boucle),
; donc DX contiendra toujours 0.
cwd
mov ax,di
; AX est le pointeur courant dnas le buffer VGA
; divise le pointeur courant par 320
mov cx,320
div cx
; DX (start_X) - reste (colonne: 0..319); AX - résultat (ligne: 0..199)
sub ax,100
; AX=AX-100, donc AX (start_Y) est maintenant dans l'intervalle -100..99
; DX est dans l'intervalle 0..319 ou 0x0000..0x013F
dec dh
; DX est maintenant dans l'intervalle 0xFF00..0x003F (-256..63)

xor bx,bx
xor si,si
; BX (temp_X)=0; SI (temp_Y)=0

; prendre le nombre maximal d'itérations
; CX contient toujours 320 ici, donc ceci est aussi le nombre maximal d'itérations
MandelLoop:
mov bp,si      ; BP = temp_Y
imul si,bx     ; SI = temp_X*temp_Y
add si,si      ; SI = SI*2 = (temp_X*temp_Y)*2
imul bx,bx     ; BX = BX^2 = temp_X^2
jo MandelBreak ; overflow?
imul bp,bp     ; BP = BP^2 = temp_Y^2
jo MandelBreak ; overflow?
add bx,bp      ; BX = BX+BP = temp_X^2 + temp_Y^2
jo MandelBreak ; overflow?
sub bx,bp      ; BX = BX-BP = temp_X^2 + temp_Y^2 - temp_Y^2 = temp_X^2
```

```
sub bx,bp       ; BX = BX-BP = temp_X^2 - temp_Y^2

; corrige l'échelle:
sar bx,6        ; BX=BX/64
add bx,dx       ; BX=BX+start_X
; maintenant temp_X = temp_X^2 - temp_Y^2 + start_X
sar si,6        ; SI=SI/64
add si,ax       ; SI=SI+start_Y
; maintenant temp_Y = (temp_X*temp_Y)*2 + start_Y

loop MandelLoop

MandelBreak:
; CX=itérations
xchg ax,cx
; AX=itérations. stocke AL dans le buffer VGA en ES:[DI]
stosb
; stosb incrémente aussi DI, donc DI pointe maintenant sur le point suivant
    dans le buffer VGA
; saute toujours, donc c'est une boucle infinie ici
jmp FillLoop
```

Algorithme:

- Change le mode vidéo à VGA 320*200 VGA, 256 couleurs. $320 * 200 = 64000$ (0xFA00).

  Chaque pixel est encodé par un octet, donc la taille du buffer est de 0xFA00 octets. Il est accédé en utilisant la paire de registres ES:DI.

  ES doit être 0xA000 ici, car c'est l'adresse du segment du buffer vidéo VGA, mais mettre 0xA000 dans ES nécessite au moins 4 octets (`PUSH 0A000h / POP ES`). Plus d'information sur le modèle de mémoire 16-bit de MS-DOS ici: 11.7 on page 1297.

  En supposant que BX vaut zéro ici, et que le Program Segment Prefix est à l'adresse zéro, l'instruction en 2 octets `LES AX,[BX]` stocke 0x20CD dans AX et 0x9FFF dans ES.

  Donc le programme commence à écrire 16 pixels (ou octets) avant le buffer vidéo. Mais c'est MS-DOS,

  Il n'y a pas de protection de la mémoire, donc l'écriture se produit tout à la fin de la mémoire conventionnelle, et en général, il n'y a rien d'important. C'est pourquoi vous voyez une bande rouge de 16 pixels de large sur le côté droit. L'image complète est décalée à gauche de 16 pixels. C'est le prix pour économiser 2 octets.

- Une boucle infinie traite chaque pixel.

  La façon la plus courante de parcourir tous les pixels de l'écran est d'utiliser deux boucles: une pour la coordonnée X, une autre pour la coordonnée Y. Mais vous devrez multiplier une coordonnée pour accéder à un octet dans le buffer vidéo VGA.

L'auteur de cette démo a décidé de faire autrement: énumérer tous les octets dans le buffer vidéo en utilisant une seule boucle au lieu de deux, et obtenir les coordonnées du point courant en utilisant une division Les coordonnées résultantes sont: X dans l'intervalle $-256..63$ et Y dans l'intervalle $-100..99$. Vous pouvez voir sur la copie d'écran que l'image est quelque peu décalée sur la droite de l'écran.

C'est parce que la surface en forme de cœur apparaît en général aux coordonnées 0,0 et elles sont décalées vers la droite. Est-ce que l'auteur aurait pu soustraire 160 de la valeur pour avoir X dans l'intervalle $-160..159$? Oui, mais l'instruction `SUB DX, 160` nécessite 4 octets, tandis que `DEC DH`—2 octets (ce qui soustrait 0x100 (256) de DX). Donc l'ensemble de l'image est décalée pour économiser 2 octets de code.

- Vérifier si le point courant est dans l'ensemble de Mandelbrot. L'algorithme est celui qui a été décrit ici.
- La boucle est organisée en utilisant l'instruction `LOOP`, qui utilise le registre CX comme compteur.

  L'auteur pourrait mettre le nombre d'itérations à une valeur spécifique, mais il ne l'a pas fait: 320 est déjà présent dans CX (il a été chargé à la ligne 35), et de toutes façons une bonne valeur d'itération maximale. Nous économisons encore un peu d'espace ici en ne rechargeant pas le registre CX avec une autre valeur.
- `IMUL` est utilisé ici au lieu de `MUL`, car nous travaillons avec des valeurs signées: gardez à l'esprit que les coordonnées 0,0 doivent être proches du centre de l'écran.

  C'est la même chose avec `SAR` (décalage arithmétique pour des valeurs signées) : c'est utilisé au lieu de `SHR`.
- Une autre idée est de simplifier le test des limites. Nous devons tester une paire de coordonnées, i.e., deux variables. Ce que je fais est de vérifier trois fois le débordement: deux opérations de mise au carré et une addition.

  En effet, nous utilisons des registres 16-bit, qui peuvent contenir des valeurs signées dans l'intervalle -32768..32767, donc si l'une des coordonnées est plus grande que 32767 lors de la multiplication signée, ce point est définitivement hors limites: nous sautons au label `MandelBreak`.
- Il y a aussi une division par 64 (instruction SAR). 64 met à l'échelle.

  Il est possible d'augmenter la valeur pour regarder de plus près, ou de la diminuer pour voir de plus loin.

• Nous sommes au label `MandelBreak`, il y a deux façons d'arriver ici: la boucle s'est terminée avec CX=0 (le point est à l'intérieur de l'ensemble de Mandelbrot) ; ou parce qu'un débordement s'est produit (CX contient toujours une valeur non zéro.) Nous écrivons la partie 8-bit basse de CX (CL) dans le buffer vidéo.

  La palette par défaut est grossière, néanmoins, 0 est noir: ainsi nous voyons du noir aux endroits où les points sont dans l'ensemble de Mandelbrot. La palette

peut être initialisée au début du programme, mais gardez à l'esprit que ceci est un programme de seulement 64 octets!

- le programme tourne en boucle infinie, car un test supplémentaire, ou une interface utilisateur quelconque nécessiterait des instructions supplémentaires.

D'autres astuces d'optimisation:

- L'instruction en 1-octet CWD est utilisée ici pour effacer DX au lieu de celle en 2-octets `XOR DX, DX` ou même celle en 3-octets `MOV DX, 0`.
- L'instruction en 1-octet `XCHG AX, CX` est utilisée ici au lieu de celle en 2-octets `MOV AX,CX`. La valeur courante de AX n'est plus nécessaire ici.
- DI (position dans le buffer vidéo) n'est pas initialisée, et contient 0xFFFE au début [51].

  C'est OK, car le programme travaille pour tout DI dans l'intervalle 0..0xFFFF éternellement, et l'utilisateur ne peut pas remarquer qu'il a commencé en dehors de l'écran (le dernier pixel d'un buffer vidéo de 320*200 est à l'adresse 0xF9FF). Donc du travail est en fait effectué en dehors des limites de l'écran.

  Autrement, il faudrait une instruction supplémentaire pour mettre DI à 0 et tester la fin du buffer vidéo.

Ma version «corrigée»

Listing 8.26 : Ma version «corrigée»

```
org 100h
mov al,13h
int 10h

; définir la palette
mov dx, 3c8h
mov al, 0
out dx, al
mov cx, 100h
inc dx
l00:
mov al, cl
shl ax, 2
out dx, al ; rouge
out dx, al ; vert
out dx, al ; bleu
loop l00

push 0a000h
pop es

xor di, di

FillLoop:
```

[51]Plus d'information sur la valeur initiale des registres: https://code.google.com/p/corkami/wiki/InitialValues#DOS

```
25 cwd
26 mov ax,di
27 mov cx,320
28 div cx
29 sub ax,100
30 sub dx,160
31
32 xor bx,bx
33 xor si,si
34
35 MandelLoop:
36 mov bp,si
37 imul si,bx
38 add si,si
39 imul bx,bx
40 jo MandelBreak
41 imul bp,bp
42 jo MandelBreak
43 add bx,bp
44 jo MandelBreak
45 sub bx,bp
46 sub bx,bp
47
48 sar bx,6
49 add bx,dx
50 sar si,6
51 add si,ax
52
53 loop MandelLoop
54
55 MandelBreak:
56 xchg ax,cx
57 stosb
58 cmp di, 0FA00h
59 jb FillLoop
60
61 ; attendre qu'une touche soit pressée
62 xor ax,ax
63 int 16h
64 ; mettre le mode vidéo texte
65 mov ax, 3
66 int 10h
67 ; sortir
68 int 20h
```

J'ai essayé de corriger toutes ces bizarreries: maintenant la palette est un dégradé de gris, le buffer vidéo est à la bonne place (lignes 19..20), l'image est dessinée au centre de l'écran (ligne 30), le programme se termine et attend qu'une touche soit pressée (lignes 58..68).

Mais maintenant c'est bien plus gros: 105 octets (ou 54 instructions) [52].

[52]Vous pouvez tester par vous-même: prenez DosBox et NASM et compilez-le avec: `nasm file.asm -fbin -o file.com`

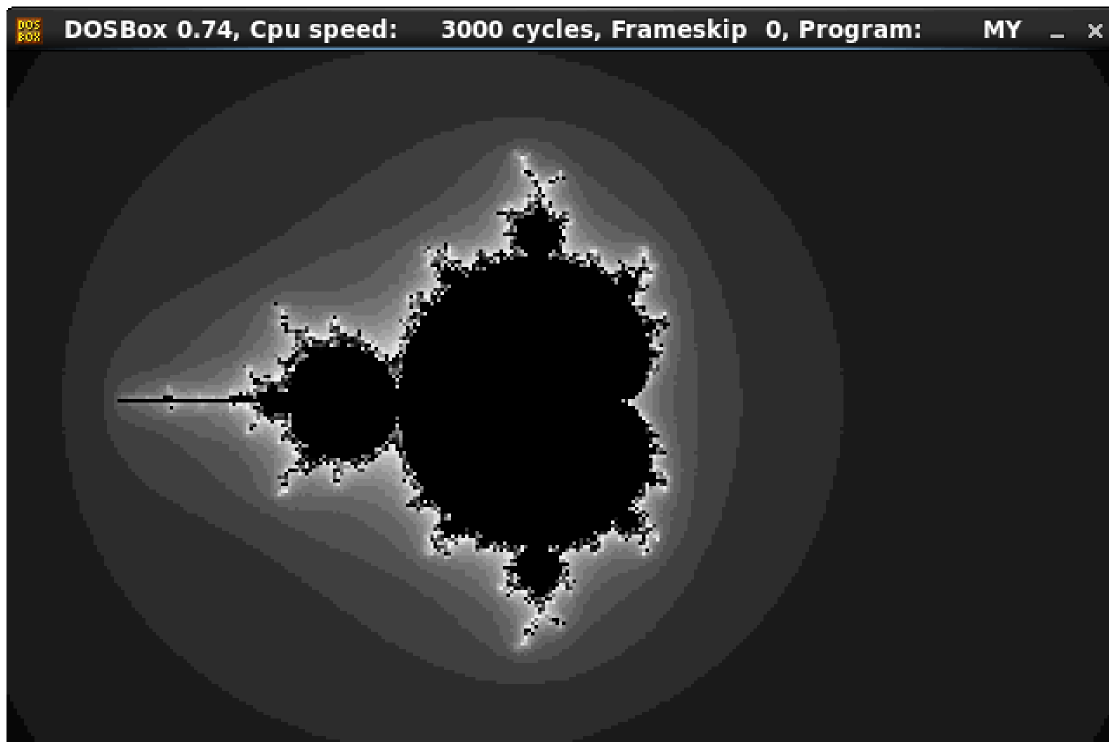


Fig. 8.18: Ma version «corrigée»

Voir aussi: petit programme C qui affiche l'ensemble de Mandelbrot en ASCII: https://people.sc.fsu.edu/~jburkardt/c_src/mandelbrot_ascii/mandelbrot_ascii.html
https://miyuki.github.io/2017/10/04/gcc-archaeology-1.html.

8.16 Un méchant bogue dans MSVCRT.DLL

Ceci est un bogue qui m'a coûté plusieurs heures de débogage.

En 2013, j'utilisais MinGW, mon projet C semblait très instable et je voyais le message d'erreur "Invalid parameter passed to C runtime function." dans le débogueur.

Le message d'erreur était aussi visible en utilisant DebugView de Sysinternals. Et mon projet n'avait pas un tel message d'erreur, ni de chaîne. Donc, j'ai commencé à chercher dans l'ensemble de Windows et l'ai trouvé dans le fichier MSVCRT.DLL (inutile de dire que j'utilisais Windows 7).

Donc le voici, le message d'erreur dans le fichier MSVCRT.DLL fourni avec Windows 7:

```
.text:6FFB69D0 OutputString    db 'Invalid parameter passed to C runtime ↙
    ↘ function.',0Ah,0
```

```
.text:6FFB69D0                                          ; DATA XREF:
    sub_6FFB6930+83
```

Où est-il référencé?

```
.text:6FFB6930 sub_6FFB6930    proc near                ;
    CODE XREF: _wfindfirst64+203FC
.text:6FFB6930                                          ; sub_6FF62563+319AD
.text:6FFB6930
.text:6FFB6930 var_2D0         = dword ptr -2D0h
.text:6FFB6930 var_244         = word ptr -244h
.text:6FFB6930 var_240         = word ptr -240h
.text:6FFB6930 var_23C         = word ptr -23Ch
.text:6FFB6930 var_238         = word ptr -238h
.text:6FFB6930 var_234         = dword ptr -234h
.text:6FFB6930 var_230         = dword ptr -230h
.text:6FFB6930 var_22C         = dword ptr -22Ch
.text:6FFB6930 var_228         = dword ptr -228h
.text:6FFB6930 var_224         = dword ptr -224h
.text:6FFB6930 var_220         = dword ptr -220h
.text:6FFB6930 var_21C         = dword ptr -21Ch
.text:6FFB6930 var_218         = dword ptr -218h
.text:6FFB6930 var_214         = word ptr -214h
.text:6FFB6930 var_210         = dword ptr -210h
.text:6FFB6930 var_20C         = dword ptr -20Ch
.text:6FFB6930 var_208         = word ptr -208h
.text:6FFB6930 var_4           = dword ptr -4
.text:6FFB6930
.text:6FFB6930                 mov     edi, edi
.text:6FFB6932                 push    ebp
.text:6FFB6933                 mov     ebp, esp
.text:6FFB6935                 sub     esp, 2D0h
.text:6FFB693B                 mov     eax, ___security_cookie
.text:6FFB6940                 xor     eax, ebp
.text:6FFB6942                 mov     [ebp+var_4], eax
.text:6FFB6945                 mov     [ebp+var_220], eax
.text:6FFB694B                 mov     [ebp+var_224], ecx
.text:6FFB6951                 mov     [ebp+var_228], edx
.text:6FFB6957                 mov     [ebp+var_22C], ebx
.text:6FFB695D                 mov     [ebp+var_230], esi
.text:6FFB6963                 mov     [ebp+var_234], edi
.text:6FFB6969                 mov     [ebp+var_208], ss
.text:6FFB696F                 mov     [ebp+var_214], cs
.text:6FFB6975                 mov     [ebp+var_238], ds
.text:6FFB697B                 mov     [ebp+var_23C], es
.text:6FFB6981                 mov     [ebp+var_240], fs
.text:6FFB6987                 mov     [ebp+var_244], gs
.text:6FFB698D                 pushf
.text:6FFB698E                 pop     [ebp+var_210]
.text:6FFB6994                 mov     eax, [ebp+4]
.text:6FFB6997                 mov     [ebp+var_218], eax
.text:6FFB699D                 lea     eax, [ebp+4]
.text:6FFB69A0                 mov     [ebp+var_2D0], 10001h
.text:6FFB69AA                 mov     [ebp+var_20C], eax
```

```
.text:6FFB69B0                  mov     eax, [eax-4]
.text:6FFB69B3                  push    offset OutputString ; "Invalid
    parameter passed to C runtime f"...
.text:6FFB69B8                  mov     [ebp+var_21C], eax
.text:6FFB69BE                  call    ds:OutputDebugStringA
.text:6FFB69C4                  mov     ecx, [ebp+var_4]
.text:6FFB69C7                  xor     ecx, ebp
.text:6FFB69C9                  call    @__security_check_cookie@4 ;
    __security_check_cookie(x)
.text:6FFB69CE                  leave
.text:6FFB69CF                  retn
.text:6FFB69CF sub_6FFB6930     endp
```

La chaîne est affichée dans le débogueur ou l'utilitaire DebugView en utilisant la fonction standard
`OutputDebugStringA()`. Comment la fonction `sub_6FFB6930` peut-elle être appelée? IDA montre au moins 290 références. En utilisant mon tracer, j'ai mis un point d'arrêt en `sub_6FFB6930` pour voir quand elle est appelée dans mon cas.

```
tracer.exe -l:1.exe bpf=msvcrt.dll!0x6FFB6930 -s

...

PID=3560|New process 1.exe
(0) msvcrt.dll!0x6ffb6930() (called from msvcrt.dll!_ftol2_sse_excpt+0↙
    ↘ x1b467 (0x759ed222))
Call stack:
return address=0x401010 (1.exe!.text+0x10), arguments in stack: 0x12ff14, 0↙
    ↘ x401010, 0x403010("asd"), 0x0, 0x12ff88, 0x4010f8
return address=0x4010f8 (1.exe!OEP+0xe3), arguments in stack: 0x12ff88, 0↙
    ↘ x4010f8, 0x1, 0x2e0ea8, 0x2e1640, 0x403000
return address=0x75b6ef3c (KERNEL32.dll!BaseThreadInitThunk+0x12), ↙
    ↘ arguments in stack: 0x12ff94, 0x75b6ef3c, 0x7ffdf000, 0x12ffd4, 0↙
    ↘ x77523688, 0x7ffdf000
return address=0x77523688 (ntdll.dll!RtlInitializeExceptionChain+0xef), ↙
    ↘ arguments in stack: 0x12ffd4, 0x77523688, 0x7ffdf000, 0x74117ec7, 0x0↙
    ↘ , 0x0
return address=0x7752365b (ntdll.dll!RtlInitializeExceptionChain+0xc2), ↙
    ↘ arguments in stack: 0x12ffec, 0x7752365b, 0x401015, 0x7ffdf000, 0x0, ↙
    ↘ 0x0
(0) msvcrt.dll!0x6ffb6930() -> 0x12f94c
PID=3560|Process 1.exe exited. ExitCode=2147483647 (0x7fffffff)
```

J'ai trouvé que mon code appelait la fonction stricmp() avec un argument à NULL. En fait, j'ai créé cet exemple en écrivant ceci:

```
#include <stdio.h>
#include <string.h>

int main()
{
        stricmp ("asd", NULL);
};
```

Si ce morceau de code est compilé en utilisant un vieux MinGW ou un vieux MSVC 6.0, il est lié avec le fichier MSVCRT.DLL. Qui, à partir de Windows 7, envoie silencieusement le message d'erreur "Invalid parameter passed to C runtime function." au débogueur et puis ne fait rien!

Voyons comment stricmp() est implémentée dans MSVCRT.DLL:

```
.text:6FF5DB38 ; Exported entry 855. _strcmpi
.text:6FF5DB38 ; Exported entry 863. _stricmp
.text:6FF5DB38
.text:6FF5DB38 ; =============== S U B R O U T I N E
    =======================================
.text:6FF5DB38
.text:6FF5DB38 ; Attributes: bp-based frame
.text:6FF5DB38
.text:6FF5DB38 ; int __cdecl strcmpi(const char *, const char *)
.text:6FF5DB38                 public _strcmpi
.text:6FF5DB38 _strcmpi        proc near               ; CODE XREF:
    LocaleEnumProc-2B
.text:6FF5DB38                                         ; LocaleEnumProc+5E
.text:6FF5DB38
.text:6FF5DB38 arg_0           = dword ptr  8
.text:6FF5DB38 arg_4           = dword ptr  0Ch
.text:6FF5DB38
.text:6FF5DB38 ; FUNCTION CHUNK AT .text:6FF68CFD SIZE 00000012 BYTES
.text:6FF5DB38 ; FUNCTION CHUNK AT .text:6FF9D20D SIZE 00000022 BYTES
.text:6FF5DB38
.text:6FF5DB38                 mov     edi, edi        ; _strcmpi
.text:6FF5DB3A                 push    ebp
.text:6FF5DB3B                 mov     ebp, esp
.text:6FF5DB3D                 push    esi
.text:6FF5DB3E                 xor     esi, esi
.text:6FF5DB40                 cmp     dword_6FFF0000, esi
.text:6FF5DB46                 jnz     loc_6FF68CFD
.text:6FF5DB4C                 cmp     [ebp+arg_0], esi  ; is arg_0==NULL?
.text:6FF5DB4F                 jz      loc_6FF9D20D
.text:6FF5DB55                 cmp     [ebp+arg_4], esi  ; is arg_0==NULL?
.text:6FF5DB58                 jz      loc_6FF9D20D
.text:6FF5DB5E                 pop     esi
.text:6FF5DB5F                 pop     ebp
.text:6FF5DB5F _strcmpi        endp ; sp-analysis failed
```

La comparaison effective des chaînes est ici:

```
.text:6FF5DB60 sub_6FF5DB60    proc near               ; CODE XREF:
    _stricmp_l+16C7F
.text:6FF5DB60                                         ; sub_6FFD19CD+229
.text:6FF5DB60
.text:6FF5DB60 arg_0           = dword ptr  8
.text:6FF5DB60 arg_4           = dword ptr  0Ch
.text:6FF5DB60
.text:6FF5DB60                 push    ebp
.text:6FF5DB61                 mov     ebp, esp
.text:6FF5DB63                 push    edi
.text:6FF5DB64                 push    esi
```

```
.text:6FF5DB65                 push    ebx
.text:6FF5DB66                 mov     esi, [ebp+arg_4]
.text:6FF5DB69                 mov     edi, [ebp+arg_0]
.text:6FF5DB6C                 mov     al, 0FFh
.text:6FF5DB6E                 mov     edi, edi
.text:6FF5DB70
.text:6FF5DB70 loc_6FF5DB70:                           ; CODE XREF:
    sub_6FF5DB60+20
.text:6FF5DB70                                         ; sub_6FF5DB60+40
.text:6FF5DB70                 or      al, al
.text:6FF5DB72                 jz      short loc_6FF5DBA6
.text:6FF5DB74                 mov     al, [esi]
.text:6FF5DB76                 add     esi, 1
.text:6FF5DB79                 mov     ah, [edi]
.text:6FF5DB7B                 add     edi, 1
.text:6FF5DB7E                 cmp     ah, al
.text:6FF5DB80                 jz      short loc_6FF5DB70
.text:6FF5DB82                 sub     al, 41h
.text:6FF5DB84                 cmp     al, 1Ah
.text:6FF5DB86                 sbb     cl, cl
.text:6FF5DB88                 and     cl, 20h
.text:6FF5DB8B                 add     al, cl
.text:6FF5DB8D                 add     al, 41h
.text:6FF5DB8F                 xchg    ah, al
.text:6FF5DB91                 sub     al, 41h
.text:6FF5DB93                 cmp     al, 1Ah
.text:6FF5DB95                 sbb     cl, cl
.text:6FF5DB97                 and     cl, 20h
.text:6FF5DB9A                 add     al, cl
.text:6FF5DB9C                 add     al, 41h
.text:6FF5DB9E                 cmp     al, ah
.text:6FF5DBA0                 jz      short loc_6FF5DB70
.text:6FF5DBA2                 sbb     al, al
.text:6FF5DBA4                 sbb     al, 0FFh
.text:6FF5DBA6
.text:6FF5DBA6 loc_6FF5DBA6:                           ; CODE XREF:
    sub_6FF5DB60+12
.text:6FF5DBA6                 movsx   eax, al
.text:6FF5DBA9                 pop     ebx
.text:6FF5DBAA                 pop     esi
.text:6FF5DBAB                 pop     edi
.text:6FF5DBAC                 leave
.text:6FF5DBAD                 retn
.text:6FF5DBAD sub_6FF5DB60    endp

.text:6FF68D0C loc_6FF68D0C:                           ; CODE XREF:
    _strcmpi+3F6F2
.text:6FF68D0C                 pop     esi
.text:6FF68D0D                 pop     ebp
.text:6FF68D0E                 retn

.text:6FF9D20D loc_6FF9D20D:                           ; CODE XREF:
    _strcmpi+17
.text:6FF9D20D                                         ; _strcmpi+20
```

```
.text:6FF9D20D                 call    near ptr _errno
.text:6FF9D212                 push    esi
.text:6FF9D213                 push    esi
.text:6FF9D214                 push    esi
.text:6FF9D215                 push    esi
.text:6FF9D216                 push    esi
.text:6FF9D217                 mov     dword ptr [eax], 16h
.text:6FF9D21D                 call    _invalid_parameter
.text:6FF9D222                 add     esp, 14h
.text:6FF9D225                 mov     eax, 7FFFFFFFh
.text:6FF9D22A                 jmp     loc_6FF68D0C
```

Maintenant la fonction `invalid_parameter()` :

```
.text:6FFB6A06                 public _invalid_parameter
.text:6FFB6A06 _invalid_parameter proc near              ; CODE XREF:
    sub_6FF5B494:loc_6FF5B618
.text:6FFB6A06                                           ;
    sub_6FF5CCFD:loc_6FF5C8A2
.text:6FFB6A06                 mov     edi, edi
.text:6FFB6A08                 push    ebp
.text:6FFB6A09                 mov     ebp, esp
.text:6FFB6A0B                 pop     ebp
.text:6FFB6A0C                 jmp     sub_6FFB6930
.text:6FFB6A0C _invalid_parameter endp
```

La fonction `invalid_parameter()` appelle finalement la fonction avec `OutputDebugStringA()` : 8.16 on page 1191.

Voyez-vous, le code de stricmp() est comme:

```
int stricmp(const char *s1, const char *s2, size_t len)
{
        if (s1==NULL || s2==NULL)
        {
                // print error message AND exit:
                return 0x7FFFFFFFh;
        };
        // do comparison
};
```

Comment se fait-il que cette erreur soit rare? Car les nouvelles versions de MSVC lient avec le fichier MSVCR120.DLL, etc. (où 120 est le numéro de version).

Jetons un coup d'œil à l'intérieur du nouveau MSVCR120.DLL de Windows 7:

```
.text:1002A0D4                 public _stricmp_l
.text:1002A0D4 _stricmp_l      proc near                 ; CODE XREF:
    _stricmp+18
.text:1002A0D4                                           ; _mbsicmp_l+47
.text:1002A0D4                                           ; DATA XREF: ...
.text:1002A0D4
.text:1002A0D4 var_10          = dword ptr -10h
.text:1002A0D4 var_8           = dword ptr -8
```

```
.text:1002A0D4 var_4           = byte ptr -4
.text:1002A0D4 arg_0           = dword ptr  8
.text:1002A0D4 arg_4           = dword ptr  0Ch
.text:1002A0D4 arg_8           = dword ptr  10h
.text:1002A0D4
.text:1002A0D4 ; FUNCTION CHUNK AT .text:1005AA7B SIZE 0000002A BYTES
.text:1002A0D4
.text:1002A0D4                 push    ebp
.text:1002A0D5                 mov     ebp, esp
.text:1002A0D7                 sub     esp, 10h
.text:1002A0DA                 lea     ecx, [ebp+var_10]
.text:1002A0DD                 push    ebx
.text:1002A0DE                 push    esi
.text:1002A0DF                 push    edi
.text:1002A0E0                 push    [ebp+arg_8]
.text:1002A0E3                 call    sub_1000F764
.text:1002A0E8                 mov     edi, [ebp+arg_0] ; arg==NULL?
.text:1002A0EB                 test    edi, edi
.text:1002A0ED                 jz      loc_1005AA7B
.text:1002A0F3                 mov     ebx, [ebp+arg_4] ; arg==NULL?
.text:1002A0F6                 test    ebx, ebx
.text:1002A0F8                 jz      loc_1005AA7B
.text:1002A0FE                 mov     eax, [ebp+var_10]
.text:1002A101                 cmp     dword ptr [eax+0A8h], 0
.text:1002A108                 jz      loc_1005AA95
.text:1002A10E                 sub     edi, ebx

...

.text:1005AA7B loc_1005AA7B:                           ; CODE XREF:
    _stricmp_l+19
.text:1005AA7B                                         ; _stricmp_l+24
.text:1005AA7B                 call    _errno
.text:1005AA80                 mov     dword ptr [eax], 16h
.text:1005AA86                 call    _invalid_parameter_noinfo
.text:1005AA8B                 mov     esi, 7FFFFFFFh
.text:1005AA90                 jmp     loc_1002A13B

...

.text:100A4670 _invalid_parameter_noinfo proc near     ; CODE XREF:
    sub_10013BEC-10F
.text:100A4670                                         ; sub_10016C0F-10F
.text:100A4670                 xor     eax, eax
.text:100A4672                 push    eax
.text:100A4673                 push    eax
.text:100A4674                 push    eax
.text:100A4675                 push    eax
.text:100A4676                 push    eax
.text:100A4677                 call    _invalid_parameter
.text:100A467C                 add     esp, 14h
.text:100A467F                 retn
.text:100A467F _invalid_parameter_noinfo endp
```

```
...

.text:100A4645 _invalid_parameter proc near           ; CODE XREF:
    _invalid_parameter(ushort const *,ushort const *,ushort const
    *,uint,uint)
.text:100A4645                                        ;
    _invalid_parameter_noinfo+7
.text:100A4645
.text:100A4645 arg_0          = dword ptr  8
.text:100A4645 arg_4          = dword ptr  0Ch
.text:100A4645 arg_8          = dword ptr  10h
.text:100A4645 arg_C          = dword ptr  14h
.text:100A4645 arg_10         = dword ptr  18h
.text:100A4645
.text:100A4645                push    ebp
.text:100A4646                mov     ebp, esp
.text:100A4648                push    dword_100E0ED8  ; Ptr
.text:100A464E                call    ds:DecodePointer
.text:100A4654                test    eax, eax
.text:100A4656                jz      short loc_100A465B
.text:100A4658                pop     ebp
.text:100A4659                jmp     eax
.text:100A465B ;
    ---------------------------------------------------------------------
.text:100A465B
.text:100A465B loc_100A465B:                          ; CODE XREF:
    _invalid_parameter+11
.text:100A465B                push    [ebp+arg_10]
.text:100A465E                push    [ebp+arg_C]
.text:100A4661                push    [ebp+arg_8]
.text:100A4664                push    [ebp+arg_4]
.text:100A4667                push    [ebp+arg_0]
.text:100A466A                call    _invoke_watson
.text:100A466F                int     3               ; Trap to Debugger
.text:100A466F _invalid_parameter endp

.text:100A469B _invoke_watson  proc near              ; CODE XREF:
    sub_1002CDB0+27068
.text:100A469B                                        ; sub_10029704+2A792
.text:100A469B                push    17h             ; ProcessorFeature
.text:100A469D                call    IsProcessorFeaturePresent
.text:100A46A2                test    eax, eax
.text:100A46A4                jz      short loc_100A46AB
.text:100A46A6                push    5
.text:100A46A8                pop     ecx
.text:100A46A9                int     29h             ; Win8:
    RtlFailFast(ecx)
.text:100A46AB ;
    ---------------------------------------------------------------------
.text:100A46AB
.text:100A46AB loc_100A46AB:                          ; CODE XREF:
    _invoke_watson+9
.text:100A46AB                push    esi
.text:100A46AC                push    1
.text:100A46AE                mov     esi, 0C0000417h
.text:100A46B3                push    esi
```

```
.text:100A46B4                push    2
.text:100A46B6                call    sub_100A4519
.text:100A46BB                push    esi              ; uExitCode
.text:100A46BC                call    __crtTerminateProcess
.text:100A46C1                add     esp, 10h
.text:100A46C4                pop     esi
.text:100A46C5                retn
.text:100A46C5 _invoke_watson  endp
```

Maintenant la fonction `invalid_parameter()` est récrite dans les nouvelles versions de MSVCR*.DLL, elle montre la boîte de dialogue, proposant de tuer le processus ou d'appeler le débogueur. Bien sûr, c'est beaucoup mieux que de retourner silencieusement. Peut-être que Microsoft a oublié de fixer MSVCRT.DLL depuis lors.

Mais comment est-ce que ça fonctionnait du temps de Windows XP? Ça ne fonctionnait pas: MSVCRT.DLL de Windows XP ne vérifiait pas si les arguments étaient NULL. Donc, sous Windows XP mon code `stricmp ("asd", NULL)` plantera, et c'est bien.

Mon hypothèse: Microsoft a mis à jour les fichiers MSVCR*.DLL (MSVCRT.DLL inclus) pour Windows 7 en ajoutant des tests de vérification partout. Toutefois, puisque MSVCRT.DLL n'était plus beaucoup utilisé depuis MSVS .NET (année 2002), il n'a pas été testé proprement et le bogue est resté. Mais des compilateurs comme MinGW peuvent toujours utiliser cette DLL.

Qu'aurais-je fait sans mes compétences en rétro-ingénierie?

Le fichier MSVCRT.DLL de Windows 8.1 a le même bogue.

8.17 Autres exemples

Un exemple à propos de Z3 et de la décompilation manuelle se trouvait ici. Il est (temporairement) déplacé là: https://sat-smt.codes.

Chapitre 9

Exemples de Reverse Engineering de format de fichier propriétaire

9.1 Chiffrement primitif avec XOR

9.1.1 Chiffrement XOR le plus simple

J'ai vu une fois un logiciel où tous les messages de débogage étaient chiffrés en utilisant XOR avec une valeur de 3. Autrement dit, les deux bits les plus bas de chaque caractères étaient inversés.

“Hello, world” devenait “Kfool/#tlqog”:

Listing 9.1 : Python

```
#!/usr/bin/python

msg="Hello, world!"

print "".join(map(lambda x: chr(ord(x)^3), msg))
```

Ceci est un chiffrement assez intéressant (ou plutôt une obfuscation), car il possède deux propriétés importantes: 1) fonction unique pour le chiffrement/déchiffrement, il suffit de l'appliquer à nouveau; 2) les caractères résultants sont aussi imprimable, donc la chaîne complète peut être utilisée dans du code source, sans caractères d'échappement.

La seconde propriété exploite le fait que tous les caractères imprimables sont organisés en lignes: 0x2x-0x7x, et lorsque vous inversez les deux bits de poids faible, le caractère est *déplacé* de 1 ou 3 caractères à droite ou à gauche, mais n'est jamais *déplacé* sur une autre ligne (peut-être non imprimable) :

```
Characters in the coded character set ascii.
---------------------------------------------------------------------
        0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
     0x C-@ C-a C-b C-c C-d C-e C-f C-g C-h TAB C-j C-k C-l RET C-n C-o
     1x C-p C-q C-r C-s C-t C-u C-v C-w C-x C-y C-z ESC C-\ C-] C-^ C-_
     2x     !   "   #   $   %   &   '   (   )   *   +   ,   -   .   /
     3x 0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?
     4x @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O
     5x P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _
     6x `   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o
     7x p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~   DEL
```

Fig. 9.1: Table ASCII 7-bit dans Emacs

…avec la seule exception du caractère 0x7F.

Par exemple, *chiffrons* les caractères de l'intervalle A-Z:

```
#!/usr/bin/python

msg="@ABCDEFGHIJKLMNO"

print "".join(map(lambda x: chr(ord(x)^3), msg))
```

Résultat: `CBA@GFEDKJIHONML`.

C'est comme si les caractères “@” et “C” avaient été échangés, ainsi que “B” et “a”.

Encore une fois, ceci est un exemple intéressant de l'exploitation des propriétés de XOR plutôt qu'un chiffrement: le même effet de *préservation de l'imprimabilité* peut être obtenu en échangeant chacun des 4 bits de poids faible, avec n'importe quelle combinaison.

9.1.2 Norton Guide: chiffrement XOR à 1 octet le plus simple possible

Norton Guide était très populaire à l'époque de MS-DOS, c'était un programme résident qui fonctionnait comme un manuel de référence hypertexte.

Les bases de données de Norton Guide étaient des fichiers avec l'extension .ng, dont le contenu avait l'air chiffré:

```
view X86.NG - Far 2.0.1807 x64 Administrator
U:\retrocomputing\MS-DOS\norton guide\X86.NG            866          372131
0000000170: 00 00 00 00 00 00 00 00 │ 00 00 18 1A B3 1A 1D 1A
0000000180: 02 1A 1A 1A 1A 1A 1A 1A │ 1A 1A 1A 1A 1A 1A 1A 1A
0000000190: 1A 1A 1A 1A FF 18 1A 1A │ 31 B5 18 1A E9 F2 18 1A
00000001A0: 69 77 19 1A B9 6B 19 1A │ 55 9A 19 1A 4E 1A 1A 1A
00000001B0: 1A 1A 1A 1A 7E 1A 1A 1A │ 1A 1A 1A 1A 74 1A 1A 1A
00000001C0: 1A 1A 1A 1A 9E 1A 1A 1A │ 1A 1A 1A 1A 95 1A 1A 1A
00000001D0: 1A 1A 1A 1A BA 1A 1A 1A │ 1A 1A 1A 1A B2 1A 1A 1A
00000001E0: 1A 1A 1A 1A 59 4A 4F 1A │ 53 74 69 6E 68 6F 79 6E
00000001F0: 73 75 74 3A 69 7F 6E 1A │ 48 7F 7D 73 69 6E 7F 68
0000000200: 69 1A 4A 68 75 6E 7F 79 │ 6E 73 75 74 36 3A 6A 68
0000000210: 73 6C 73 76 7F 7D 7F 1A │ 5F 62 79 7F 6A 6E 73 75
0000000220: 74 69 1A 5B 7E 7E 68 7F │ 69 69 73 74 7D 3A 77 75
0000000230: 7E 7F 69 1A 55 6A 79 75 │ 7E 7F 69 1A 1A 18 1A 51
0000000240: 1A 19 1A 12 1A 1A 1A 1A │ 1A 1A 1A 1A 1A 1A 1A 1A
0000000250: 1A 1A 1A 1A 1A 1A 1A 21 │ A4 19 1A 99 A8 1E 1A 3E
0000000260: 1A 1A 1A 1A 1A 1A 1A 2E │ 1A 1A 1A 1A 1A 1A 1A 50
0000000270: 1A 1A 1A 1A 1A 1A 1A 5C │ 4A 4F 1A 53 74 69 6E 68
0000000280: 6F 79 6E 73 75 74 3A 69 │ 7F 6E 1A 48 7F 7D 73 69
0000000290: 6E 7F 68 69 36 3A 7E 7B │ 6E 7B 3A 6E 63 6A 7F 69
00000002A0: 1A 1A 18 1A 33 1A 18 1A │ 1E 1A 1A 1A 1A 1A 1A 1A
00000002B0: 1A 1A 1A 1A 1A 1A 1A 1A │ 1A 1A 1A 1A AC C6 1E 1A
00000002C0: 02 1A 1A 1A 1A 1A 1A 1A │ 32 1A 1A 1A 1A 1A 1A 1A
00000002D0: 57 57 42 1A 53 74 69 6E │ 68 6F 79 6E 73 75 74 3A
00000002E0: 69 7F 6E 1A 1A 1A 1A 8B │ 09 99 1A 1A 1A E5 E5 E5
00000002F0: E5 E5 E5 1A 1A 1A 1A 1A │ 1A 1A 1A 1A 1A 1A 1A 08
0000000300: 19 8A 0C 1A 1A 2E 19 62 │ 01 1A 1A 4D 19 CC 07 1A
0000000310: 1A 63 19 72 3A 1A 1A 84 │ 19 39 3E 1A 1A A9 19 1A
0000000320: 33 1A 1A A7 19 7F 37 1A │ 1A CB 19 67 28 1A 1A 18
1      2      3      4      5Print 6      7Prev  8Goto  9Video 10
```

Fig. 9.2: Aspect très typique

Pourquoi pensons-nous qu'il est chiffré mais pas compressé?

Nous voyons que l'octet 0x1A (ressemblant à « → ») est très fréquent, ça ne serait pas possible dans un fichier compressé.

Nous voyons aussi de longues parties constituées seulement de lettres latines, et qui ressemble à des chaînes de caractères dans un langage inconnu.

Puisque l'octet 0x1A revient si souvent, nous pouvons essayer de décrypter le fichier, en supposant qu'il est chiffré avec le chiffrement XOR le plus simple.

Si nous appliquons un XOR avec la constante 0x1A à chaque octet dans Hiew, nous voyons des chaînes de texte familières en anglais:

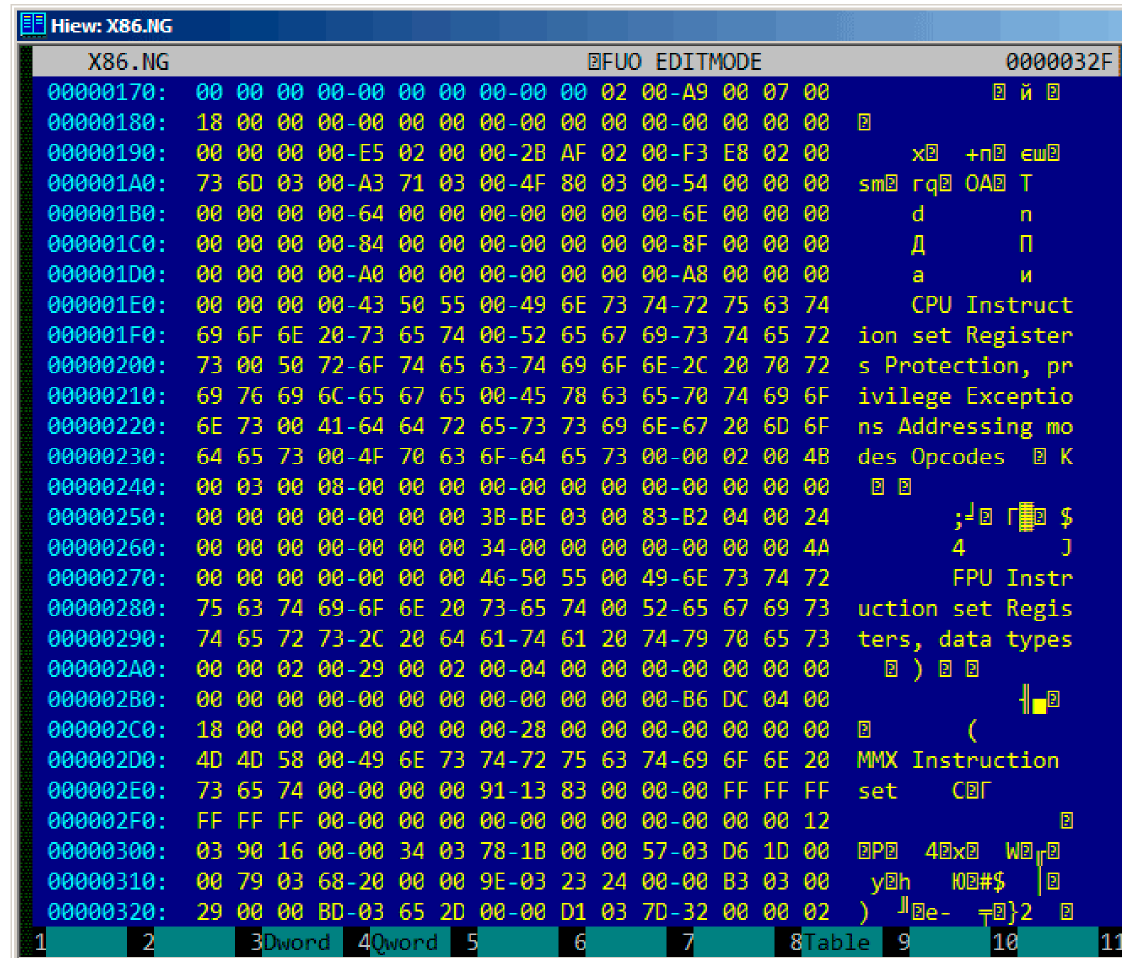


Fig. 9.3: XOR dans Hiew avec 0x1A

Le chiffrement XOR avec un seul octet constant est la méthode de chiffrement la plus simple, que l'on rencontre néanmoins parfois.

Maintenant, nous comprenons pourquoi l'octet 0x1A revenait si souvent: parce qu'il y a beaucoup d'octets à zéro et qu'ils sont remplacés par 0x1A dans la forme chiffrée.

Mais la constante pourrait être différente. Dans ce cas, nous pourrions essayer chaque constante dans l'intervalle 0..255 et chercher quelque chose de familier dans le fichier déchiffré. 256 n'est pas si grand.

Plus d'informations sur le format de fichier de Norton Guide: http://www.davep.org/norton-guides/file-format/.

Entropie

Une propriété très importante de tels systèmes de chiffrement est que l'entropie des blocs chiffrés/déchiffrés est la même.

Voici mon analyse faite dans Wolfram Mathematica 10.

Listing 9.2 : Wolfram Mathematica 10

```
In[1]:= input = BinaryReadList["X86.NG"];

In[2]:= Entropy[2, input] // N
Out[2]= 5.62724

In[3]:= decrypted = Map[BitXor[#, 16^^1A] &, input];

In[4]:= Export["X86_decrypted.NG", decrypted, "Binary"];

In[5]:= Entropy[2, decrypted] // N
Out[5]= 5.62724

In[6]:= Entropy[2, ExampleData[{"Text", "ShakespearesSonnets"}]] // N
Out[6]= 4.42366
```

Ici, nous chargeons le fichier, obtenons son entropie, le déchiffrons, le sauvons et obtenons à nouveau son entropie (la même!).

Mathematica fourni également quelques textes en langue anglaise bien connus pour analyse.

Nous obtenons ainsi l'entropie de sonnets de Shakespeare, et elle est proche de l'entropie du fichier que nous venons d'analyser.

Le fichier que nous avons analysé consiste en des phrases en langue anglaise, qui sont proches du langage de Shakespeare.

Et le texte en langue anglaise XOR-é possède la même entropie.

Toutefois, ceci n'est pas vrai lorsque le fichier est XOR-é avec un pattern de plus d'un octet.

Le fichier qui vient d'être analysé peut être téléchargé ici: http://beginners.re/examples/norton_guide/X86.NG.

Encore un mot sur la base de l'entropie

Wolfram Mathematica calcule l'entropie avec une base e (base des logarithmes naturels), et l'utilitaire[1]UNIX *ent* utilise une base 2.

[1]http://www.fourmilab.ch/random/

Donc, nous avons mis explicitement une base 2 dans la commande `Entropy`, donc Mathematica nous donne le même résultat que l'utilitaire *ent*.

9.1.3 Chiffrement le plus simple possible avec un XOR de 4-octets

Si un pattern plus long était utilisé, comme un pattern de 4 octets, ça serait facile à repérer.

Par exemple, voici le début du fichier kernel32.dll (version 32-bit de Windows Server 2008) :

Fig. 9.4: Fichier original

Ici, il est «chiffré» avec une clef de 4-octet:

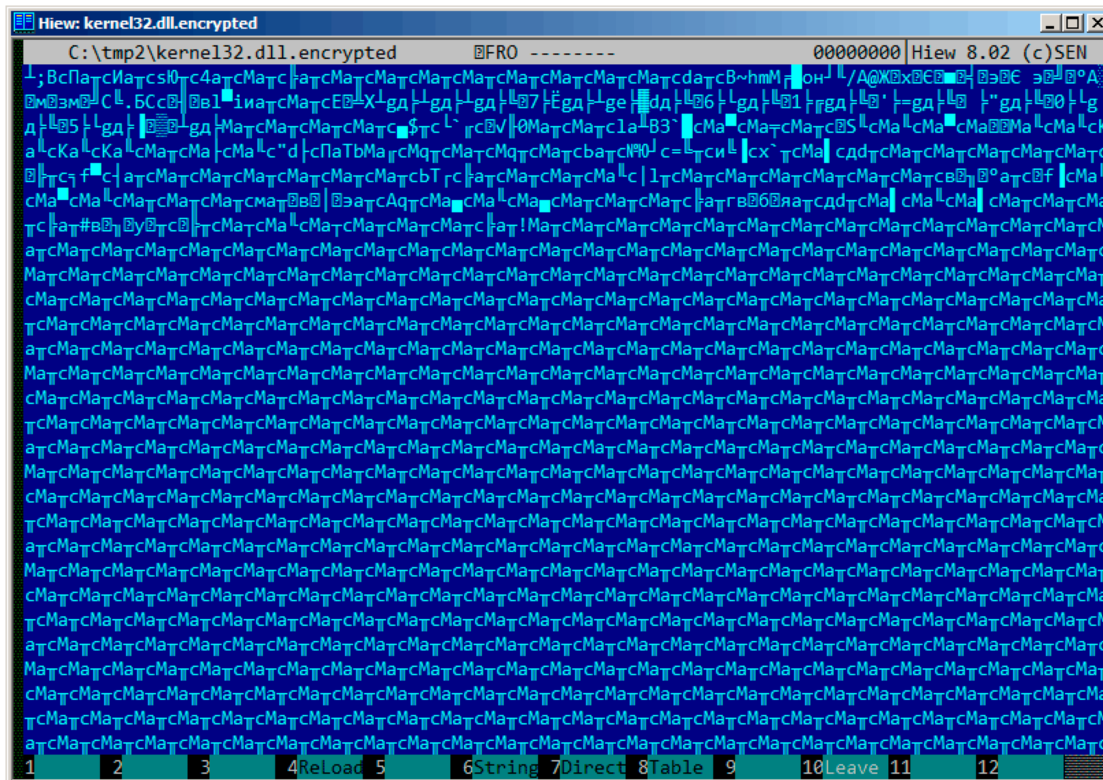


Fig. 9.5: Fichier «chiffré»

Il est facile de repérer les 4 symboles récurrents.

En effet, l'entête d'un fichier PE comporte de longues zones de zéro, ce qui explique que la clef devient visible.

Voici le début d'un entête PE au format hexadécimal:

```
Hiew: kernel32.dll
   C:\tmp2\kernel32.dll                                  FRO --------                 PE .7DD60290
.7DD600E0:  00 00 00 00-00 00 00 00-50 45 00 00-4C 01 04 00
.7DD600F0:  85 9A 15 53-00 00 00 00-00 00 00 00-E0 00 02 21
.7DD60100:  0B 01 09 00-00 00 0D 00-00 00 03 00-00 00 00 00
.7DD60110:  93 32 01 00-00 00 01 00-00 00 0D 00-00 00 D6 7D
.7DD60120:  00 00 01 00-00 00 01 00-06 00 01 00-06 00 01 00
.7DD60130:  06 00 01 00-00 00 00 00-00 00 11 00-00 00 01 00
.7DD60140:  AE 05 11 00-03 00 40 01-00 00 04 00-00 10 00 00
.7DD60150:  00 00 10 00-00 10 00 00-00 00 00 00-10 00 00 00
.7DD60160:  70 FF 0B 00-B1 A9 00 00-24 A9 0C 00-F4 01 00 00
.7DD60170:  00 00 0F 00-28 05 00 00-00 00 00 00-00 00 00 00
.7DD60180:  00 00 00 00-00 00 00 00-00 00 10 00-9C AD 00 00
.7DD60190:  34 07 0D 00-38 00 00 00-00 00 00 00-00 00 00 00
.7DD601A0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.7DD601B0:  10 35 08 00-40 00 00 00-00 00 00 00-00 00 00 00
.7DD601C0:  00 00 01 00-F0 0D 00 00-00 00 00 00-00 00 00 00
.7DD601D0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.7DD601E0:  2E 74 65 78-74 00 00 00-96 07 0C 00-00 00 01 00   .text
.7DD601F0:  00 00 0D 00-00 00 01 00-00 00 00 00-00 00 00 00
.7DD60200:  00 00 00 00-20 00 00 60-2E 64 61 74-61 00 00 00          `.data
.7DD60210:  0C 10 00 00-00 00 0E 00-00 00 01 00-00 00 0E 00
.7DD60220:  00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 C0
.7DD60230:  2E 72 73 72-63 00 00 00-28 05 00 00-00 00 0F 00   .rsrc
.7DD60240:  00 00 01 00-00 00 0F 00-00 00 00 00-00 00 00 00
.7DD60250:  00 00 00 00-40 00 00 40-2E 72 65 6C-6F 63 00 00       @  @.reloc
.7DD60260:  9C AD 00 00-00 00 10 00-00 00 01 00-00 00 10 00
.7DD60270:  00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 42
.7DD60280:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.7DD60290:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
1Global 2FilBlk 3CryBlk 4ReLoad 5       6String 7Direct 8Table 9       10Leave 11
```

Fig. 9.6: Entête PE

Le voici «chiffré » :

```
Hiew: kernel32.dll.encrypted
   C:\tmp2\kernel32.dll.encrypted          FRO --------                00000290
 000000E0:  8C 61 D2 63-8C 61 D2 63-DC 24 D2 63-C0 60 D6 63
 000000F0:  09 FB C7 30-8C 61 D2 63-8C 61 D2 63-6C 61 D0 42
 00000100:  87 60 DB 63-8C 61 DF 63-8C 61 D1 63-8C 61 D2 63
 00000110:  1F 53 D3 63-8C 61 D3 63-8C 61 DF 63-8C 61 04 1E
 00000120:  8C 61 D3 63-8C 61 D3 63-8A 61 D3 63-8A 61 D3 63
 00000130:  8A 61 D3 63-8C 61 D2 63-8C 61 C3 63-8C 61 D3 63
 00000140:  22 64 C3 63-8F 61 92 62-8C 61 D6 63-8C 71 D2 63
 00000150:  8C 61 C2 63-8C 71 D2 63-8C 61 D2 63-9C 61 D2 63
 00000160:  FC 9E D9 63-3D C8 D2 63-A8 C8 DE 63-78 60 D2 63
 00000170:  8C 61 DD 63-A4 64 D2 63-8C 61 D2 63-8C 61 D2 63
 00000180:  8C 61 D2 63-8C 61 D2 63-8C 61 C2 63-10 CC D2 63
 00000190:  B8 66 DF 63-B4 61 D2 63-8C 61 D2 63-8C 61 D2 63
 000001A0:  8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-8C 61 D2 63
 000001B0:  9C 54 DA 63-CC 61 D2 63-8C 61 D2 63-8C 61 D2 63
 000001C0:  8C 61 D3 63-7C 6C D2 63-8C 61 D2 63-8C 61 D2 63
 000001D0:  8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-8C 61 D2 63
 000001E0:  A2 15 B7 1B-F8 61 D2 63-1A 66 DE 63-8C 61 D3 63
 000001F0:  8C 61 DF 63-8C 61 D3 63-8C 61 D2 63-8C 61 D2 63
 00000200:  8C 61 D2 63-AC 61 D2 03-A2 05 B3 17-ED 61 D2 63
 00000210:  80 71 D2 63-8C 61 DC 63-8C 61 D3 63-8C 61 DC 63
 00000220:  8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-CC 61 D2 A3
 00000230:  A2 13 A1 11-EF 61 D2 63-A4 64 D2 63-8C 61 DD 63
 00000240:  8C 61 D3 63-8C 61 DD 63-8C 61 D2 63-8C 61 D2 63
 00000250:  8C 61 D2 63-CC 61 D2 23-A2 13 B7 0F-E3 02 D2 63
 00000260:  10 CC D2 63-8C 61 C2 63-8C 61 D3 63-8C 61 C2 63
 00000270:  8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-CC 61 D2 21
 00000280:  8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-8C 61 D2 63
 00000290:  8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-8C 61 D2 63
1Global 2FilBlk 3CryBlk 4ReLoad 5       6String 7Direct 8Table 9       10Leave 11
```

Fig. 9.7: Entête PE «chiffré »

Il est facile de repérer que la clef est la séquence de 4 octets suivant: 8C 61 D2 63.

Avec cette information, c'est facile de déchiffrer le fichier entier.

Il est important de garder à l'esprit ces propriétés importantes des fichiers PE: 1) l'entête PE comporte de nombreuses zones remplies de zéro; 2) toutes les sections PE sont complétées avec des zéros jusqu'à une limite de page (4096 octets), donc il y a d'habitude de longues zones à zéro après chaque section.

Quelques autres formats de fichier contiennent de longues zones de zéro.

C'est typique des fichiers utilisés par les scientifiques et les ingénieurs logiciels.

Pour ceux qui veulent inspecter ces fichiers eux-même, ils sont téléchargeables ici: http://beginners.re/examples/XOR_4byte/.

Exercice

- http://challenges.re/50

9.1.4 Chiffrement simple utilisant un masque XOR

J'ai trouvé un vieux jeu de fiction interactif en plongeant profondément dans *if-archive*[2] :

```
The New Castle v3.5 - Text/Adventure Game
in the style of the original Infocom (tm)
type games, Zork, Collosal Cave (Adventure),
etc.  Can you solve the mystery of the
abandoned castle?
Shareware from Software Customization.
Software Customization [ASP] Version 3.5 Feb. 2000
```

Il est téléchargeable ici.

Il y a un fichier à l'intérieur (appelé *castle.dbf*) qui est visiblement chiffré, mais pas avec un vrai algorithme de crypto, qui n'est pas non plus compressé, il s'agit plutôt de quelque chose de plus simple. Je ne vais même pas mesurer le niveau d'entropie (9.2 on page 1225) du fichier, car je suis sûr qu'il est bas. Voici à quoi il ressemble dans Midnight Commander:

/home/dennis/P/RE-book/decrypt_dat_file/castle.dbf

Fig. 9.8: Fichier chiffré dans Midnight Commander

Le fichier chiffré peut être téléchargé ici: .

[2] http://www.ifarchive.org/

Sera-t-il possible de le décrypter sans accéder au programme, en utilisant juste ce fichier?

Il y a clairement un pattern visible de chaînes répétées. Si un simple chiffrement avec un masque XOR a été appliqué, une répétition de telles chaînes en est une signature notable, car, il y avait probablement de longues suites (lacunes[3]) d'octets à zéro, qui, à tour de rôle, sont présentes dans de nombreux fichiers exécutables, tout comme dans des fichiers de données binaires.

Ici, je vais afficher le début du fichier en utilisant l'utilitaire UNIX *xxd* :

```
...

0000030: 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e 1d 61  .a.c.w.iubgv.~.a
0000040: 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e 7a 02  z..rn..}}c~wf.z.
0000050: 75 50 02 4a 31 71 31 33 5c 27 08 5c 51 74 3e 39  uP.J1q13\'.\Qt>9
0000060: 50 2e 28 72 24 4b 38 21 4c 09 37 38 3b 51 41 2d  P.(r$K8!L.78;QA-
0000070: 1c 3c 37 5d 27 5a 1c 7c 6a 10 14 68 77 08 6d 1a  .<7]'Z.|j..hw.m.

0000080: 6a 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e 1d  j.a.c.w.iubgv.~.
0000090: 61 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e 7a  az..rn..}}c~wf.z
00000a0: 02 75 50 64 02 74 71 66 76 19 63 08 13 17 74 7d  .uPd.tqfv.c...t}
00000b0: 6b 19 63 6d 72 66 0e 79 73 1f 09 75 71 6f 05 04  k.cmrf.ys..uqo..
00000c0: 7f 1c 7a 65 08 6e 0e 12 7c 6a 10 14 68 77 08 6d  ..ze.n..|j..hw.m

00000d0: 1a 6a 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e  .j.a.c.w.iubgv.~
00000e0: 1d 61 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e  .az..rn..}}c~wf.
00000f0: 7a 02 75 50 01 4a 3b 71 2d 38 56 34 5b 13 40 3c  z.uP.J;q-8V4[.@<
0000100: 3c 3f 19 26 3b 3b 2a 0e 35 26 4d 42 26 71 26 4b  <?.&;;*.5&MB&q&K
0000110: 04 2b 54 3f 65 40 2b 4f 40 28 39 10 5b 2e 77 45  .+T?e@+O@(9.[.wE

0000120: 28 54 75 09 61 0d 63 0f 77 14 69 75 62 67 76 01  (Tu.a.c.w.iubgv.
0000130: 7e 1d 61 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66  ~.az..rn..}}c~wf
0000140: 1e 7a 02 75 50 02 4a 31 71 15 3e 58 27 47 44 17  .z.uP.J1q.>X'GD.
0000150: 3f 33 24 4e 30 6c 72 66 0e 79 73 1f 09 75 71 6f  ?3$N0lrf.ys..uqo
0000160: 05 04 7f 1c 7a 65 08 6e 0e 12 7c 6a 10 14 68 77  ....ze.n..|j..hw

...
```

Concentrons-nous sur la chaîne visible `iubgv` se répétant. En regardant ce dump, nous voyons clairement que la période de l'occurrence de la chaîne est 0x51 ou 81. La taille du fichier est 1658961, et est divisible par 81 (et il y a donc 20481 blocs).

Maintenant, je vais utiliser Mathematica pour l'analyse, y a-t-il des blocs de 81 octets se répètant dans le fichier? Je vais séparer le fichier d'entrée en blocs de 81 octets et ensuite utiliser la fonction *Tally[]*[4] qui compte simplement combien de fois un élément était présent dans la liste en entrée. La sortie de Tally n'est pas triée, donc je vais ajouter la fonction *Sort[]* pour trier le nombre d'occurrences par ordre décroissant.

```
input = BinaryReadList["/home/dennis/.../castle.dbf"];
```

[3] Comme dans https://en.wikipedia.org/wiki/Lacuna_(manuscripts)
[4] https://reference.wolfram.com/language/ref/Tally.html

```
blocks = Partition[input, 81];

stat = Sort[Tally[blocks], #1[[2]] > #2[[2]] &]
```

Et voici la sortie:

```
{{{80, 103, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, 125, 107,
   25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, 5, 4,
   127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, 8,
   109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118,
   1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126,
   119, 102, 30, 122, 2, 117}, 1739},
{{80, 100, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116,
   125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113,
   111, 5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20,
   104, 119, 8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117,
   98, 103, 118, 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125,
   125, 99, 126, 119, 102, 30, 122, 2, 117}, 1422},
{{80, 101, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116,
   125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113,
   111, 5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20,
   104, 119, 8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117,
   98, 103, 118, 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125,
   125, 99, 126, 119, 102, 30, 122, 2, 117}, 1012},
{{80, 120, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116,
   125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113,
   111, 5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20,
   104, 119, 8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117,
   98, 103, 118, 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125,
   125, 99, 126, 119, 102, 30, 122, 2, 117}, 377},

...

{{80, 2, 74, 49, 113, 21, 62, 88, 39, 71, 68, 23, 63, 51, 36, 78, 48,
   108, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, 5, 4, 127, 28,
   122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, 8, 109, 26,
   106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126,
   29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102,
   30, 122, 2, 117}, 1},
{{80, 1, 74, 59, 113, 45, 56, 86, 52, 91, 19, 64, 60, 60, 63,
   25, 38, 59, 59, 42, 14, 53, 38, 77, 66, 38, 113, 38, 75, 4, 43, 84,
    63, 101, 64, 43, 79, 64, 40, 57, 16, 91, 46, 119, 69, 40, 84, 117,
    9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126, 29,
   97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102, 30,
   122, 2, 117}, 1},
{{80, 2, 74, 49, 113, 49, 51, 92, 39, 8, 92, 81, 116, 62, 57,
   80, 46, 40, 114, 36, 75, 56, 33, 76, 9, 55, 56, 59, 81, 65, 45, 28,
    60, 55, 93, 39, 90, 28, 124, 106, 16, 20, 104, 119, 8, 109, 26,
   106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126,
   29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102,
   30, 122, 2, 117}, 1}}
```

La sortie de Tally est une liste de paires, chaque paire a un bloc de 81 octets et le nombre de fois qu'il apparaît dans le fichier. Nous voyons que le bloc le plus fréquent est le premier, il est apparu 1739 fois. Le second apparaît 1422 fois. Puis les autres: 1012 fois, 377 fois, etc. Les blocs de 81 octets qui ne sont apparus qu'une fois sont à la fin de la sortie.

Essayons de comparer ces blocs. Le premier et le second. Y a-t-il une fonction dans Mathematica qui compare les listes/tableaux? Certainement qu'il y en a une, mais dans un but didactique, je vais utiliser l'opération XOR pour la comparaison. En effet: si les octets dans deux tableaux d'entrée sont identiques, le résultat du XOR est 0. Si ils sont différents, le résultat sera différent de zéro.

Comparons le premier bloc (qui apparaît 1739 fois) et le second (qui apparaît 1422 fois) :

```
In[]:= BitXor[stat[[1]][[1]], stat[[2]][[1]]]
Out[]= {0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Ils ne diffèrent que par le second octet.

Comparons le second bloc (qui apparaît 1422 fois) et le troisième (qui apparaît 1012 fois) :

```
In[]:= BitXor[stat[[2]][[1]], stat[[3]][[1]]]
Out[]= {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Ils ne diffèrent également que par le second octet.

Quoiqu'il en soit, essayons d'utiliser le bloc qui apparaît le plus comme une clef XOR et essayons de déchiffrer les quatre premiers blocs de 81 octets dans le fichier:

```
In[]:= key = stat[[1]][[1]]
Out[]= {80, 103, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, \
125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, \
5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, \
8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, \
1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, \
102, 30, 122, 2, 117}

In[]:= ToASCII[val_] := If[val == 0, " ", FromCharacterCode[val, "↙
    ↘ PrintableASCII"]]

In[]:= DecryptBlockASCII[blk_] := Map[ToASCII[#] &, BitXor[key, blk]]

In[]:= DecryptBlockASCII[blocks[[1]]]
Out[]= {" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
```

```
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " "}

In[]:= DecryptBlockASCII[blocks[[2]]]
Out[]= {" ", "e", "H", "E", " ", "W", "E", "E", "D", " ", "O", \
"F", " ", "C", "R", "I", "M", "E", " ", "B", "E", "A", "R", "S", " ", \
"B", "I", "T", "T", "E", "R", " ", "F", "R", "U", "I", "T", "?", \
" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", \
" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", \
" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", \
" "}

In[]:= DecryptBlockASCII[blocks[[3]]]
Out[]= {" ", "?", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
"}

In[]:= DecryptBlockASCII[blocks[[4]]]
Out[]= {" ", "f", "H", "O", " ", "K", "N", "O", "W", "S", " ", \
"W", "H", "A", "T", " ", "E", "V", "I", "L", " ", "L", "U", "R", "K", \
"S", " ", "I", "N", " ", "T", "H", "E", " ", "H", "E", "A", "R", "T", \
"S", " ", "O", "F", " ", "M", "E", "N", "?", " ", " ", " ", " ", \
" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", \
" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", \
" "}
```

(J’ai remplacé les caractères non imprimables par «? ».)

Donc nous voyons que le premier et le troisième blocs sont vides (ou presque vide), mais le second et le quatrième comportent clairement des mots/phrases en anglais. Ils semble que notre hypothèse à propos de la clef soit correct (au moins en partie). Cala signifie que le bloc de 81 octets qui apparaît le plus souvent dans le fichier peut être trouvé à des endroits comportant des séries d’octets à zéro ou quelque chose comme ça.

Essayons de déchiffrer le fichier entier:

```
DecryptBlock[blk_] := BitXor[key, blk]

decrypted = Map[DecryptBlock[#] &, blocks];

BinaryWrite["/home/dennis/.../tmp", Flatten[decrypted]]

Close["/home/dennis/.../tmp"]
```

```
RE-book/decrypt_dat_file/tmp                                   4011/1620K       0%
...................................................................eHE.WEED.OF
TTER.FRUIT....................................................................
...........................................fHO.KNOWS.WHAT.EVIL.LURKS.IN.THE.HE
...............................eHE.sHADOW.KNOWS...............................
..............................................................................
.......x.HAVE.THE.HEART.OF.A.CHILD............................................
P.IT.IN.A.GLASS.JAR.ON.MY.DESK................................................
................................................................uEVERON.......
....................................................fHERE.THE.sHADOW.LIES.....
..............................................................................
............................pLL.POSITIONING.IS.relative.AND.NOT.absolute......
..............................................................................
....eHIS.IS.A.KLUDGE.TO.MAKE.THIS.STUPID.THING.WORK...........................
.........................................................................cELAX
Y..............cLOCK.tICKS.AWAY...............................................
.................................................uEBUGGING.pROGRAMS.IS.FUN...s
AD
K.WALLS..        .............................................................
..............pND.FROM.WITHIN.THE.TOMB.OF.THE.UNDEAD..VAMPIRES.BEGAN.THEIR.FEA
..EORTURED.CRIES.RANG.OUT.....tASTES.GREAT..lESS.FILLING......................
.......................................................................bUDDENL
RAITHLIKE.FIGURE.APPEARS.BEFORE.YOU..SEEMING.TO............WLOAT.IN.THE.AIR...
WFUL.VOICE.HE.SAYS...aLAS..THE.VERY............_ATURE.OF.THE.WORLD.HAS.CHANGED
ON.CANNOT.BE.FOUND...aLL...........\UST.NOW.PASS.AWAY....rAISING.HIS.OAKEN.STA
.HE.FADES.INTO.........EHE.SPREADING.DARKNESS...iN.HIS.PLACE.APPEARS.A.TASTEFU
GN.........CEADING............................................................
..............................................................................
```

Fig. 9.9: Fichier déchiffré dans Midnight Commander, 1er essai

Ceci ressemble a des sortes de phrases en anglais d'un jeu, mais quelque chose ne va pas. Tout d'abord, la casse est inversée: les phrases et certains mots commence avec une minuscule, tandis que d'autres caractères sont en majuscule. De plus, certaines phrases commencent avec une mauvaise lettre. Regardez la toute première phrase: «eHE WEED OF CRIME BEARS BITTER FRUIT». Que signifie «eHE»? Ne devrait-on pas avoir «tHE» ici? Est-il possible que notre clef de déchiffrement ait un mauvais octet à cet endroit?

Regardons à nouveau le second bloc dans le fichier, la clef et le résultat décrypté:

```
In[]:= blocks[[2]]
Out[]= {80, 2, 74, 49, 113, 49, 51, 92, 39, 8, 92, 81, 116, 62, \
57, 80, 46, 40, 114, 36, 75, 56, 33, 76, 9, 55, 56, 59, 81, 65, 45, \
28, 60, 55, 93, 39, 90, 28, 124, 106, 16, 20, 104, 119, 8, 109, 26, \
106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126, 29, \
97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102, 30, \
122, 2, 117}

In[]:= key
Out[]= {80, 103, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, \
125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, \
5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, \
```

```
8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, \
1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, \
102, 30, 122, 2, 117}

In[]:= BitXor[key, blocks[[2]]]
Out[]= {0, 101, 72, 69, 0, 87, 69, 69, 68, 0, 79, 70, 0, 67, 82, \
73, 77, 69, 0, 66, 69, 65, 82, 83, 0, 66, 73, 84, 84, 69, 82, 0, 70, \
82, 85, 73, 84, 14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0}
```

L'octet chiffré est 2, l'octet de la clef est 103, $2 \oplus 103 = 101$ et 101 est le code ASCII du caractère «e». A quoi devrait être égal l'octet de la clef, afin que le code ASCII résultant soit 116 (pour le caractère «t»)? $2 \oplus 116 = 118$, mettons 118 comme second octet de la clef …

```
key = {80, 118, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, 125,
  107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, 5,
  4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, 8,
   109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118,
  1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119,
   102, 30, 122, 2, 117}
```

…et déchiffrons le fichier à nouveau.

```
/home/dennis/P/RE-book/decrypt_dat_file/tmp                        4011/1620K          0%
..................................................................................tHE.WEED.OF
.CRIME.BEARS.BITTER.FRUIT.....................................................................
..........................................................wHO.KNOWS.WHAT.EVIL.LURKS.IN.THE.HE
ARTS.OF.MEN...................................tHE.sHADOW.KNOWS...............................
..............................................................................................
......................i.HAVE.THE.HEART.OF.A.CHILD............................................
..........i.KEEP.IT.IN.A.GLASS.JAR.ON.MY.DESK.................................................
...............................................................................dEVERON.......
...................................................................wHERE.THE.sHADOW.LIES.....
..............................................................................................
.............................................aLL.POSITIONING.IS.relative.AND.NOT.absolute.....
..............................................................................................
...................tHIS.IS.A.KLUDGE.TO.MAKE.THIS.STUPID.THING.WORK...........................
...........................................................................................rELAX
..fRIDAY.IS.ONLY..............cLOCK.tICKS.AWAY................................................
.................................................................dEBUGGING.pROGRAMS.IS.FUN...s
O.IS.RUNNING.HEAD
FIRST.INTO.BRICK.WALLS..        .............................................................
.............................aND.FROM.WITHIN.THE.TOMB.OF.THE.UNDEAD..VAMPIRES.BEGAN.THEIR.FEA
ST.AS...........TORTURED.CRIES.RANG.OUT.....tASTES.GREAT..lESS.FILLING.......................
.........................................................................................sUDDENL
Y.A.SINISTER..WRAITHLIKE.FIGURE.APPEARS.BEFORE.YOU..SEEMING.TO...........FLOAT.IN.THE.AIR...
iN.A.LOW..SORROWFUL.VOICE.HE.SAYS...aLAS..THE.VERY...........NATURE.OF.THE.WORLD.HAS.CHANGED
..AND.THE.DUNGEON.CANNOT.BE.FOUND...aLL...........MUST.NOW.PASS.AWAY....rAISING.HIS.OAKEN.STA
FF.IN.FAIRWELL..HE.FADES.INTO.........THE.SPREADING.DARKNESS...iN.HIS.PLACE.APPEARS.A.TASTEFU
LLY.LETTERED.SIGN.........READING.............................................................
```

Fig. 9.10: Fichier déchiffré dans Midnight Commander, 2nd essai

Ouah, maintenant, la grammaire est correcte, toutes les phrases commencent avec

une lettre correcte. Mais encore, l'inversion de la casse est suspecte. Pourquoi est-ce que le développeur les aurait écrites de cette façon? Peut-être que notre clef est toujours incorrecte?

En regardant la table ASCII, nous pouvons remarquer que les codes des lettres majuscules et des minuscules ne diffèrent que d'un bit (6ème bit en partant de 1, 0b100000) :

```
Characters in the coded character set ascii.
---------------------------------------------------------------------
      0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
  0x C-@ C-a C-b C-c C-d C-e C-f C-g C-h TAB C-j C-k C-l RET C-n C-o
  1x C-p C-q C-r C-s C-t C-u C-v C-w C-x C-y C-z ESC C-\ C-] C-^ C-_
  2x      !   "   #   $   %   &   '   (   )   *   +   ,   -   .   /
  3x  0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?
  4x  @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O
  5x  P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^
  6x  `   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o
  7x  p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~  DEL
```

Fig. 9.11: table ASCII 7-bit dans Emacs

Cet octet avec seul le 6ème bit mis est 32 au format décimal. Mais 32 est le code ASCII de l'espace!

En effet, on peut changer la casse juste en XOR-ant le code ASCII d'un caractère avec 32 (plus à ce sujet: 3.19.3 on page 691).

Est-ce possible que les parties vides dans le fichier ne soient pas des octets à zéro, mais plutôt des espaces? Modifions notre clef XOR encore une fois (je vais appliquer Un XOR avec 32 à chaque octet de la clef) :

```
(* "32" is scalar and "key" is vector, but that's OK *)

In[]:= key3 = BitXor[32, key]
Out[]= {112, 86, 34, 84, 81, 70, 86, 57, 67, 40, 51, 55, 84, 93, 75, \
57, 67, 77, 82, 70, 46, 89, 83, 63, 41, 85, 81, 79, 37, 36, 95, 60, \
90, 69, 40, 78, 46, 50, 92, 74, 48, 52, 72, 87, 40, 77, 58, 74, 41, \
65, 45, 67, 47, 87, 52, 73, 85, 66, 71, 86, 33, 94, 61, 65, 90, 49, \
47, 82, 78, 35, 37, 93, 93, 67, 94, 87, 70, 62, 90, 34, 85}

In[]:= DecryptBlock[blk_] := BitXor[key3, blk]
```

Déchiffrons à nouveau le fichier d'entrée:

```
/home/dennis/P/RE-book/decrypt_dat_file/tmp3
 1                                                                                   The
                            2
in the hearts of men?                                        The Shadow knows!
 2                                                                                   I h
                            I keep it in a glass jar on my desk.
                                                             Deveron:
 Where the Shadow lies.                                                              1
                            All positioning is RELATIVE and not ABSOLUTE.
                                                             This is a kludge to make this
 1                                                                                   Re
                            1
 (So is running head-first into brick walls!!)               2
 And from within the tomb of the undead, vampires began their feast as               to
g!"                          10
hlike figure appears before you, seeming to                  float in the air.  In a low, s
 nature of the world has changed, and the dungeon cannot be found.  All              mus
well, he fades into         the spreading darkness.  In his place appears a tasteful

                  INITIALIZATION FAILURE
                            The darkness becomes all encompassing, and your vision fa
                                                             Lick My User Port!!!
 1                                                                                   So
CRATCH Paper.               1
hem you were playing GAMES all day...                        1
 Keep it up and we'll both go out for a beer.                                        1
                            No, odd addresses don't occur on the South side of the st
                                                             Did you really expect me to re
 1                                                                                   In
                            1
```

Fig. 9.12: Fichier déchiffré dans Midnight Commander, essai final

(Le fichier déchiffré est disponible ici.)

Ceci est indiscutablement un fichier source correct. Oh, et nous voyons des nombres au début de chaque bloc. Ça doit être la source de notre clef XOR erronée. Il semble que le bloc de 81 octets le plus fréquent dans le fichier soit un bloc rempli avec des espaces et contenant le caractère «1» à la place du second octet. En effet, d'une façon ou d'une autre, de nombreux blocs sont entrelacés avec celui-ci.

Peut-être est-ce une sorte de remplissage pour les phrases/messages courts? D'autres blocs de 81 octets sont aussi remplis avec des blocs d'espaces, mais avec un chiffre différent, ainsi, ils ne diffèrent que du second octet.

C'est tout! Maintenant nous pouvons écrire un utilitaire pour chiffrer à nouveau le fichier, et peut-être le modifier avant.

Le fichier notebook de Mathematica est téléchargeable ici.

Résumé: un tel chiffrement avec XOR n'est pas robuste du tout. Le développeur du jeu escomptait, probablement, empêcher les joueurs de chercher des informations sur le jeu, mais rien de plus sérieux. Néanmoins, un tel chiffrement est très populaire du fait de sa simplicité et de nombreux rétro-ingénieurs sont traditionnellement familier avec.

9.1.5 Chiffrement simple utilisant un masque XOR, cas II

J'ai un autre fichier chiffré, qui est clairement chiffré avec quelque chose de simple, comme un XOR:

```
/home/dennis/tmp/cipher.txt                                    0x00000000
00000000 DD D2 0F 70 | 1C E7 9E 8D | E9 EC AC 3D | 61 5A 15 95   .p.     =aZ.
00000010 5C F5 D3 0D | 70 38 E7 94 | DF F2 E2 BC | 76 34 61 0F  \  .p8      v4a.
00000020 98 5D FC D9 | 01 26 2A FD | 82 DF E9 E2 | BB 33 61 7B   ]  .&*      3a{
00000030 14 D9 45 F8 | C5 01 3D 20 | FD 95 96 EB | E4 BC 7A 61  . E  .=       za
00000040 61 1B 8F 54 | 9D AA 54 20 | 20 E1 DB 8B | ED EC BC 33  a. T  T        3
00000050 61 7C 15 8D | 11 F9 CE 47 | 22 2A FE 8E | 9A EB F7 EF  al. .  G"*
00000060 39 22 71 1B | 8A 58 FF CE | 52 70 38 E7 | 9E 91 A5 EB  9"q. X  Rp8
00000070 AA 76 36 73 | 09 D9 44 E0 | 80 40 3C 23 | AF 95 96 E2   v6s. D  @<#
00000080 EB BB 7A 61 | 65 1B 8A 11 | E3 C5 40 24 | 2A EB F6 F5    zae. .  @$*
00000090 E4 F7 EF 22 | 29 77 5A 9B | 43 F5 C1 4A | 36 2E FC 8F     ")wZ C  J6.
000000A0 DF F1 E2 AD | 3A 24 3C 5A | B0 11 E3 D4 | 4E 3F 2B AF      :$<Z .  N?+
000000B0 8E 8F EA ED | EF 22 29 77 | 5A 91 54 F1 | D2 55 38 62       ")wZ T  U8b
000000C0 FD 8E 98 A5 | E2 A1 32 61 | 62 13 9A 5A | F5 C4 01 25        2ab. Z  .%
000000D0 3F AF 8F 97 | E0 8E C5 25 | 35 7B 19 92 | 11 E7 C8 48  ?      %5{. .  H
000000E0 33 27 AF 94 | 8A F7 A3 B9 | 3F 32 7B 0E | 96 43 B0 C8  3'      ?2{. C
000000F0 40 34 6F E3 | 9E 99 F1 A3 | AD 33 29 7B | 14 9D 11 F8  @4o      3){. .
00000100 C9 4C 70 3B | E7 9E DF EB | EA A8 3E 35 | 32 18 9C 57   Lp;      >52. W
00000110 FF D2 44 7E | 6F C6 8F DF | F2 E2 BC 76 | 20 1F 70 9F    D~o      v .p
00000120 58 FE C5 0D | 70 3B E7 92 | 9C EE A3 BF | 3F 24 71 1F  X  .p;     ?$q.
00000130 D9 5E F6 80 | 56 3F 20 EB | D7 DF E7 F6 | A3 34 2E 67   ^  V?        4.g
00000140 09 D4 59 F5 | C1 45 35 2B | A3 DB 90 E3 | A3 BB 3E 24  . Y  E5+      >$
00000150 32 09 96 43 | E4 80 56 38 | 26 EC 93 DF | EC F0 EF 3D  2. C  V8&       =
00000160 2F 7D 0D 97 | 11 F1 D3 2C | 5A 2E AF D9 | AF E0 ED AE  /}. .  ,Z.
00000170 38 26 32 16 | 98 46 E9 C5 | 53 7E 6D AF | B1 8A F6 F7  8&2. F  S~m
00000180 EF 23 2F 76 | 1F 8B 11 E4 | C8 44 70 27 | EA 9A 9B A5   #/v. .  Dp'.
00000190 F4 AE 25 61 | 73 5A 9B 43 | FF C1 45 70 | 3C E6 97 89    %asZ C  Ep<
000001A0 E0 F1 EF 34 | 20 7C 1E D9 | 5F F5 C1 53 | 3C 36 82 F1     4 |. _  S<6
000001B0 9E EB A3 A6 | 38 22 7A 5A | 98 52 E2 CF | 52 23 61 AF     8"zZ R  R#a
000001C0 D9 AB EA A3 | 85 37 2C 77 | 09 D9 7C FF | D2 55 39 22   .  7,w. |  U9"
000001D0 EA 89 D3 A5 | CE E1 04 6F | 51 54 AA 1F | BC 80 47 22         .oQT .  G"
000001E0 20 E2 DB 97 | EC F0 EF 30 | 33 7B 1F 97 | 55 E3 80 4E        03{. U  N
000001F0 36 6F FB 93 | 9A 88 89 8C | 78 02 3C 32 | D7 1D B2 80  6o       x.<2 .
```

Fig. 9.13: Fichier chiffré dans Midnight Commander

Le fichier chiffré peut être téléchargé ici.

L'utilitaire Linux *ent* indique environ ~ 7.5 bits par octet, et ceci est un haut niveau d'entropie (9.2 on page 1225), proche de celui de fichiers compressés ou chiffrés correctement. Mais encore, nous distinguons clairement quelques patterns, il y a quelques blocs avec une taille de 17 octets, et nous pouvons voir des sortes d'échelles, se décalant d'un octet à chaque ligne de 16 octets.

On sait aussi que le texte clair est en anglais.

Maintenant, supposons que ce morceau de texte est chiffré par un simple XOR avec une clef de 17 octets.

J’ai essayé de repérer des blocs de 17 octets se répétant avec Mathematica, comme je l’ai fait dans l’exemple précédant (9.1.4 on page 1209) :

Listing 9.3 : Mathematica

```
In[]:=input = BinaryReadList["/home/dennis/tmp/cipher.txt"];

In[]:=blocks = Partition[input, 17];

In[]:=Sort[Tally[blocks], #1[[2]] > #2[[2]] &]

Out[]:={{{248,128,88,63,58,175,159,154,232,226,161,50,97,127,3,217,80},1},
{{226,207,67,60,42,226,219,150,246,163,166,56,97,101,18,144,82},1},
{{228,128,79,49,59,250,137,154,165,236,169,118,53,122,31,217,65},1},
{{252,217,1,39,39,238,143,223,241,235,170,91,75,119,2,152,82},1},
{{244,204,88,112,59,234,151,147,165,238,170,118,49,126,27,144,95},1},
{{241,196,78,112,54,224,142,223,242,236,186,58,37,50,17,144,95},1},
{{176,201,71,112,56,230,143,151,234,246,187,118,44,125,8,156,17},1},
...
{{255,206,82,112,56,231,158,145,165,235,170,118,54,115,9,217,68},1},
{{249,206,71,34,42,254,142,154,235,247,239,57,34,113,27,138,88},1},
{{157,170,84,32,32,225,219,139,237,236,188,51,97,124,21,141,17},1},
{{248,197,1,61,32,253,149,150,235,228,188,122,97,97,27,143,84},1},
{{252,217,1,38,42,253,130,223,233,226,187,51,97,123,20,217,69},1},
{{245,211,13,112,56,231,148,223,242,226,188,118,52,97,15,152,93},1},
{{221,210,15,112,28,231,158,141,233,236,172,61,97,90,21,149,92},1}}
```

Pas de chance, chaque bloc de 17 octets est unique dans le fichier, et n’apparaît donc qu’une fois. Peut-être n’y a-t-il pas de zone de 17 octets à zéro, ou de zone contenant seulement des espaces. C’est possible toutefois: de telles séries d’espace peuvent être absentes dans des textes composés rigoureusement.

La première idée est d’essayer toutes les clefs de 17 octets possible et trouver celles qui donnent un résultat lisible après déchiffrement. La force brute n’est pas une option, car il y a 256^{17} clefs possible (~10^{40}), c’est beaucoup trop. Mais il y a une bonne nouvelle: qui a dit que nous devons tester la clef de 17 octets en entier, pourquoi ne pas teste chaque octet séparémment? C’est possible en effet.

Maintenant, l’algorithme est:

- essayer chacun des 25 octets pour le premier octet de la clef;
- déchiffrer le 1er octet de chaque bloc de 17 octets du fichier;
- est-ce que tous les octets déchiffrés sont imprimable? garder un œil dessus;
- faire de même pour chacun des 17 octets de la clef.

J’ai écrit le script Python suivant pour essayer cette idée:

Listing 9.4 : Python script

```
each_Nth_byte=[""]*KEY_LEN

content=read_file(sys.argv[1])
# split input by 17-byte chunks:
```

```
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks:
    for i in range(KEY_LEN):
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN):
    print "N=", N
    possible_keys=[]
    for i in range(256):
        tmp_key=chr(i)*len(each_Nth_byte[N])
        tmp=xor_strings(tmp_key,each_Nth_byte[N])
        # are all characters in tmp[] are printable?
        if is_string_printable(tmp)==False:
            continue
        possible_keys.append(i)
    print possible_keys, "len=", len(possible_keys)
```

(La version complète du code source est ici.)

Voici sa sortie:

```
N= 0
[144, 145, 151] len= 3
N= 1
[160, 161] len= 2
N= 2
[32, 33, 38] len= 3
N= 3
[80, 81, 87] len= 3
N= 4
[78, 79] len= 2
N= 5
[142, 143] len= 2
N= 6
[250, 251] len= 2
N= 7
[254, 255] len= 2
N= 8
[130, 132, 133] len= 3
N= 9
[130, 131] len= 2
N= 10
[206, 207] len= 2
N= 11
[81, 86, 87] len= 3
N= 12
[64, 65] len= 2
N= 13
[18, 19] len= 2
N= 14
[122, 123] len= 2
N= 15
[248, 249] len= 2
```

```
N= 16
[48, 49] len= 2
```

Donc, il y a 2 ou 3 octets possible pour chaque octet de l clef de 17 octets. C'est mieux que 256 octets pour chaque octet, mais encore beaucoup trop. Il y a environ 1 million de clefs possible:

Listing 9.5 : Mathematica

```
In[]:= 3*2*3*3*2*2*2*2*3*2*2*3*2*2*2*2*2
Out[]= 995328
```

Il est possible de les vérifier toutes, mais alors nous devons vérifier visuellement si le texte déchiffré à l'air d'un texte en anglais.

Prenons en compte le fait que nous avons à faire avec 1) un langage naturel 2) de l'anglais. Les langages naturels ont quelques caractéristiques statistiques importantes. Tout d'abord, le ponctuation et la longueur des mots. Quelle est la longueur moyenne des mots en anglais? Comptons les espaces dans quelques textes bien connus en anglais avec Mathematica.

Voici le fichier texte de «The Complete Works of William Shakespeare» provenant de la bibliothèque Gutenberg.

Listing 9.6 : Mathematica

```
In[]:= input = BinaryReadList["/home/dennis/tmp/pg100.txt"];

In[]:= Tally[input]
Out[]= {{239, 1}, {187, 1}, {191, 1}, {84, 39878}, {104,
  218875}, {101, 406157}, {32, 1285884}, {80, 12038}, {114,
  209907}, {111, 282560}, {106, 2788}, {99, 67194}, {116,
  291243}, {71, 11261}, {117, 115225}, {110, 216805}, {98,
  46768}, {103, 57328}, {69, 42703}, {66, 15450}, {107, 29345}, {102,
  69103}, {67, 21526}, {109, 95890}, {112, 46849}, {108, 146532}, {87,
   16508}, {115, 215605}, {105, 199130}, {97, 245509}, {83,
  34082}, {44, 83315}, {121, 85549}, {13, 124787}, {10, 124787}, {119,
   73155}, {100, 134216}, {118, 34077}, {46, 78216}, {89, 9128}, {45,
  8150}, {76, 23919}, {42, 73}, {79, 33268}, {82, 29040}, {73,
  55893}, {72, 18486}, {68, 15726}, {58, 1843}, {65, 44560}, {49,
  982}, {50, 373}, {48, 325}, {91, 2076}, {35, 3}, {93, 2068}, {74,
  2071}, {57, 966}, {52, 107}, {70, 11770}, {85, 14169}, {78,
  27393}, {75, 6206}, {77, 15887}, {120, 4681}, {33, 8840}, {60,
  468}, {86, 3587}, {51, 343}, {88, 608}, {40, 643}, {41, 644}, {62,
  440}, {39, 31077}, {34, 488}, {59, 17199}, {126, 1}, {95, 71}, {113,
   2414}, {81, 1179}, {63, 10476}, {47, 48}, {55, 45}, {54, 73}, {64,
  3}, {53, 94}, {56, 47}, {122, 1098}, {90, 532}, {124, 33}, {38,
  21}, {96, 1}, {125, 2}, {37, 1}, {36, 2}}

In[]:= Length[input]/1285884 // N
Out[]= 4.34712
```

Il y a 1285884 espaces dans l'ensemble du fichier, et la fréquence de l'occurrence des espaces est de 1 par ~4.3 caractères.

Maintenant voici Alice’s Adventures in Wonderland, par Lewis Carroll de la même bibliothèque:

Listing 9.7 : Mathematica

```
In[]:= input = BinaryReadList["/home/dennis/tmp/pg11.txt"];

In[]:= Tally[input]
Out[]= {{239, 1}, {187, 1}, {191, 1}, {80, 172}, {114, 6398}, {111,
  9243}, {106, 222}, {101, 15082}, {99, 2815}, {116, 11629}, {32,
  27964}, {71, 193}, {117, 3867}, {110, 7869}, {98, 1621}, {103,
  2750}, {39, 2885}, {115, 6980}, {65, 721}, {108, 5053}, {105,
  7802}, {100, 5227}, {118, 911}, {87, 256}, {97, 9081}, {44,
  2566}, {121, 2442}, {76, 158}, {119, 2696}, {67, 185}, {13,
  3735}, {10, 3735}, {84, 571}, {104, 7580}, {66, 125}, {107,
  1202}, {102, 2248}, {109, 2245}, {46, 1206}, {89, 142}, {112,
  1796}, {45, 744}, {58, 255}, {68, 242}, {74, 13}, {50, 12}, {53,
  13}, {48, 22}, {56, 10}, {91, 4}, {69, 313}, {35, 1}, {49, 68}, {93,
   4}, {82, 212}, {77, 222}, {57, 11}, {52, 10}, {42, 88}, {83,
  288}, {79, 234}, {70, 134}, {72, 309}, {73, 831}, {85, 111}, {78,
  182}, {75, 88}, {86, 52}, {51, 13}, {63, 202}, {40, 76}, {41,
  76}, {59, 194}, {33, 451}, {113, 135}, {120, 170}, {90, 1}, {122,
  79}, {34, 135}, {95, 4}, {81, 85}, {88, 6}, {47, 24}, {55, 6}, {54,
  7}, {37, 1}, {64, 2}, {36, 2}}

In[]:= Length[input]/27964 // N
Out[]= 5.99049
```

Le résultat est différent, sans soute à cause d’un formatage des textes différents (indentation ou remplissage).

Ok, donc supposons que la fréquence moyenne de l’espace en anglais est de 1 espace tous les 4..7 caractères.

Maintenant, encore une bonne nouvelle: nous pouvons mesurer la fréquence des espaces au fur et à mesure du déchiffrement de notre fichier. Maintenant je compte les espaces dans chaque *slice* et jette les clefs de 1 octets qui produise un résultat avec un nombre d’espaces trop petit (ou trop grand, mais c’est presque impossible avec une si petite clef) :

Listing 9.8 : Python script

```
each_Nth_byte=[""]*KEY_LEN

content=read_file(sys.argv[1])
# split input by 17-byte chunks:
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks:
    for i in range(KEY_LEN):
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN):
    print "N=", N
    possible_keys=[]
```

```
    for i in range(256):
        tmp_key=chr(i)*len(each_Nth_byte[N])
        tmp=xor_strings(tmp_key,each_Nth_byte[N])
        # are all characters in tmp[] are printable?
        if is_string_printable(tmp)==False:
            continue
        # count spaces in decrypted buffer:
        spaces=tmp.count(' ')
        if spaces==0:
            continue
        spaces_ratio=len(tmp)/spaces
        if spaces_ratio<4:
            continue
        if spaces_ratio>7:
            continue
        possible_keys.append(i)
    print possible_keys, "len=", len(possible_keys)
```

(La version complète du code source se trouve ici.)

Ceci nous donne un seul octet possible pour chaque octet de la clef:

```
N= 0
[144] len= 1
N= 1
[160] len= 1
N= 2
[33] len= 1
N= 3
[80] len= 1
N= 4
[79] len= 1
N= 5
[143] len= 1
N= 6
[251] len= 1
N= 7
[255] len= 1
N= 8
[133] len= 1
N= 9
[131] len= 1
N= 10
[207] len= 1
N= 11
[86] len= 1
N= 12
[65] len= 1
N= 13
[18] len= 1
N= 14
[122] len= 1
N= 15
[249] len= 1
```

```
N= 16
[49] len= 1
```

Vérifions cette clef dans Mathematica:

Listing 9.9 : Mathematica

```
In[]:= input = BinaryReadList["/home/dennis/tmp/cipher.txt"];

In[]:= blocks = Partition[input, 17];

In[]:= key = {144, 160, 33, 80, 79, 143, 251, 255, 133, 131, 207, 86, 65, ↙
    ↘ 18, 122, 249, 49};

In[]:= EncryptBlock[blk_] := BitXor[key, blk]

In[]:= encrypted = Map[EncryptBlock[#] &, blocks];

In[]:= BinaryWrite["/home/dennis/tmp/plain2.txt", Flatten[encrypted]]

In[]:= Close["/home/dennis/tmp/plain2.txt"]
```

Et le texte brut est:

```
Mr. Sherlock Holmes, who was usually very late in the mornings, save
upon those not infrequent occasions when he was up all night, was seated
at the breakfast table. I stood upon the hearth-rug and picked up the
stick which our visitor had left behind him the night before. It was a
fine, thick piece of wood, bulbous-headed, of the sort which is known as
a "Penang lawyer." Just under the head was a broad silver band nearly
an inch across. "To James Mortimer, M.R.C.S., from his friends of the
C.C.H.," was engraved upon it, with the date "1884." It was just such a
stick as the old-fashioned family practitioner used to carry--dignified,
solid, and reassuring.

"Well, Watson, what do you make of it?"

Holmes was sitting with his back to me, and I had given him no sign of
my occupation.

...
```

(La version complète de ce texte se trouve ici.)

Le texte semble correct. Oui, j'ai créé cet exemple de toutes pièces et j'ai choisi un texte très connu de Conan Doyle, mais c'est très proche de ce que j'ai eu à faire il y a quelques temps.

Autres idées à envisager

Si nous échouions avec le comptage des espaces, il y a d'autres idées à essayer:

- Prenons en considération le fait que les lettres minuscules sont plus fréquentes que celles en majuscule.

- Analyse des fréquences.
- Il y a aussi une bonne technique pour détecter le langage d'un texte: les trigrammes. Chaque langage possède des triplets de lettres fréquences, qui peuvent être «the» et «tha» en anglais. En lire plus à ce sujet: N-Gram-Based Text Categorization, http://code.activestate.com/recipes/326576/. Fait suffisamment intéressant, la détection des trigrammes peut être utilisée lorsque vous décryptez un texte chiffré progressivement, comme dans cet exemple (yous devez juste tester les 3 caractères décryptez adjacents).

  Pour les systèmes non-latin encodés en UTF-8, les choses peuvent être plus simples. Par exemple, les textes en russe encodés en UTF-8 ont chaque octet intercalé avec des octets 0xD0/0xD1. C'est parce que les caractères cyrilliques sont situés dans le 4ème bloc de la table Unicode. D'autres systèmes d'écriture on leurs propres blocs.

9.1.6 Devoir

Un ancien jeu d'aventure en texte pour MS-DOS, développé à la fin des années 1980. Pour cacher les informations du jeu aux joueurs, les fichiers de données sont, le plus probablement, XORé avec quelque chose: https://beginners.re/homework/XOR_crypto_1/destiny.zip. Essayez d'y rentrer…

9.2 Information avec l'entropie

> Entropy: The quantitative measure of disorder, which in turn relates to the thermodynamic functions, temperature, and heat.
>
> Dictionaire: Applied Math for Engineers and Scientists

Par soucis de simplification, je dirais que l'entropie est une mesure, d'à quel point des données peuvent être compressées. Par exemple, il est généralement impossible de compresser un fichier archive déjà compressé, donc il a une entropie importante. D'un autre coté, 1MiB d'octet à zéro peut être compressé en un tout petit fichier. En effet, en français, un million de zéro peut être simplement décrit par "le fichier résultant est un million d'octets à zéro". Les fichiers compressés sont en général une liste d'instructions destinées au dé-compresseur, comme ceci: "mettre 1000 zéros, puis l'octet 0x23, puis l'octet 0x45, puis un bloc d'une taille de 10 octets que nous avons vu 500 octets avant, etc."

Les textes écrits en langage naturel peuvent aussi être fortement compressés, car le langage naturel a beaucoup de redondance (autrement, une petite typo conduirait toujours à une incompréhension, comme un bit inversé dans un fichier archive rend la décompression presque impossible), certains mots sont utilisés très souvent, etc. Dans le discours courant, il est possible de supprimer jusqu'à la moitié des mots et il est toujours compréhensible.

Le code pour les CPUs peut aussi être compressé, car certaines instructions ISA sont utilisées plus souvent que d'autres. En x86, les instructions les plus utilisées sont MOV/PUSH/CALL (5.11.2 on page 951).

La compression de données et le chiffrement tendent à produire des résultats avec une très haute entropie. Un bon PRNG produit aussi des données qui ne peuvent pas être compressées (il est possible de mesurer leur qualité par ce moyen).

Donc, autrement dit, la mesure de l'entropie peut aider à tester le contenu de bloc de données inconnues.

9.2.1 Analyse de l'entropie dans Mathematica

(Cette partie est parue initialement sur mon blog le 13 mai 2015. Quelques discussions: https://news.ycombinator.com/item?id=9545276.)

Il est possible de découper un fichier par blocs, de calculer l'entropie de chacun d'eux et de dessiner un graphe. J'ai fais ceci avec Wolfram Mathematica à titre de démonstration et voici le code source (Mathematica 10) :

```
(* loading the file *)
input=BinaryReadList["file.bin"];

(* setting block sizes *)
BlockSize=4096;BlockSizeToShow=256;

(* slice blocks by 4k *)
blocks=Partition[input,BlockSize];

(* how many blocks we've got? *)
Length[blocks]

(* calculate entropy for each block. 2 in Entropy[] (base) is set with the ↙
    ↘ intention so Entropy[]
function will produce the same results as Linux ent utility does *)
entropies=Map[N[Entropy[2,#]]&,blocks];

(* helper functions *)
fBlockToShow[input_,offset_]:=Take[input,{1+offset,1+offset+BlockSizeToShow↙
    ↘ }]
fToASCII[val_]:=FromCharacterCode[val,"PrintableASCII"]
fToHex[val_]:=IntegerString[val,16]
fPutASCIIWindow[data_]:=Framed[Grid[Partition[Map[fToASCII,data],16]]]
fPutHexWindow[data_]:=Framed[Grid[Partition[Map[fToHex,data],16],Alignment↙
    ↘ ->Right]]

(* that will be the main knob here *)
{Slider[Dynamic[offset],{0,Length[input]-BlockSize,BlockSize}],Dynamic[↙
    ↘ BaseForm[offset,16]]}

(* main UI part *)
Dynamic[{ListLinePlot[entropies,GridLines->{{-1,offset/BlockSize,1}},↙
    ↘ Filling->Axis,AxesLabel->{"offset","entropy"}],
CurrentBlock=fBlockToShow[input,offset];
```

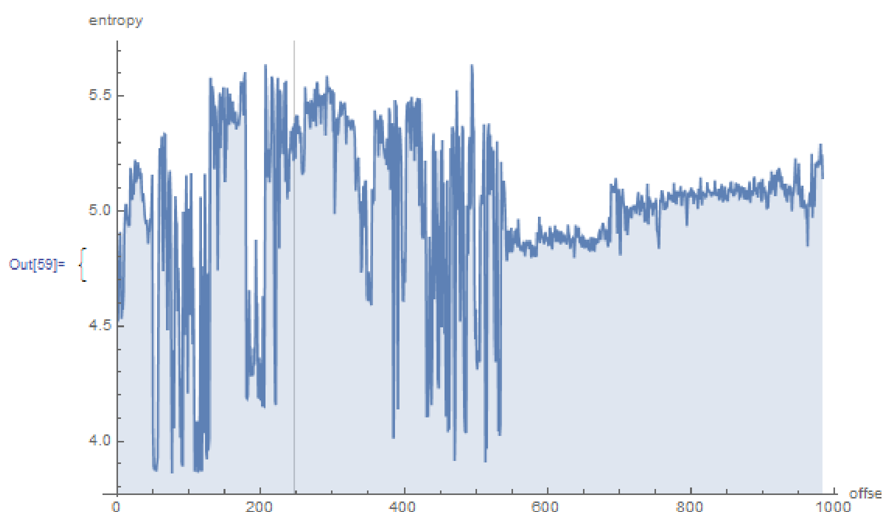
```
fPutHexWindow[CurrentBlock],
fPutASCIIWindow[CurrentBlock]}]
```

Base de données GeoIP de FAI

Commençons avec le fichier GeoIP (qui assigne un FAI au bloc d'adresses IP). Ce fichier binaire *GeoIPISP.dat* a plusieurs tables (qui sont peut-être les intervalles d'adresses IP) plus quelques blobs de texte à la fin du fichier (contenant les noms des FAI).

Lorsque je le charge dans Mathematica, je vois ceci:

```
In[68]:= (* that will be the main knob here *)
         {Slider[Dynamic[offset], {0, Length[input] - BlockSize, BlockSize}]

Out[68]= {                    , f7000₁₆}

In[59]:= (* main UI part *)
         Dynamic[{ListLinePlot[entropies, GridLines → {{-1, offset / BlockSiz
           CurrentBlock = fBlockToShow[input, offset];
           fPutHexWindow[CurrentBlock], fPutASCIIWindow[CurrentBlock]}]

Out[59]= {
```

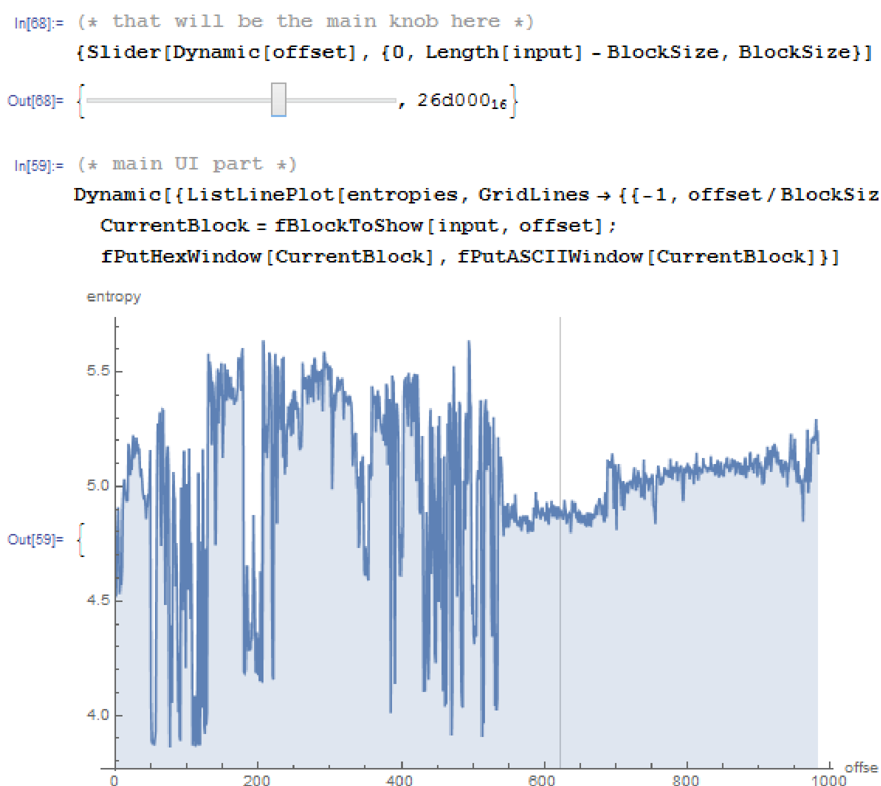



```
 1 ee  1 0 76 eb 5 0 17 c0  5 0 f3 de  5 0      □ □ □ □ v □ □ □ □ □ □ □ □ □ □ □
76 eb  5 0  3 ee 1 0  4 ee  1 0  5 ee  1 0      v □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
76 eb  5 0 aa 6c 6 0 fd 2c  4 0 64 59 14 0      v □ □ □ □ l □ □ □ , □ □ d Y □ □
 7 ee  1 0  a ee 1 0 64 59 14 0  8 ee  1 0      □ □ □ □   □ □ □ d Y □ □ □ □ □ □
64 59 14 0  9 ee 1 0 f0 3d  6 0 4c d3  6 0      d Y □ □ □ □ □ □ □ = □ □ L □ □ □
 b ee  1 0  e ee 1 0  c ee  1 0  d ee  1 0      □ □ □ □ □ □ □ □ □ □ □ □   □ □ □
4e bc  6 0 17 45 6 0 fd 2c  4 0 b6 ed  4 0      N □ □ □ □ E □ □ □ , □ □ □ □ □ □
 f ee  1 0 10 ee 1 0 fd 2c  4 0 f3 a3  5 0      □ □ □ □ □ □ □ □ □ , □ □ □ □ □ □
8d df  5 0  2 dc 6 0 12 ee  1 0 2c ee  1 0 ,    □ □ □ □ □ □ □ □ □ □ □ □ , □ □ □ }
13 ee  1 0 1d ee 1 0 40 14  6 0 14 ee  1 0      □ □ □ □ □ □ □ □ @ □ □ □ □ □ □ □
15 ee  1 0 19 ee 1 0 40 14  6 0 16 ee  1 0      □ □ □ □ □ □ □ □ @ □ □ □ □ □ □ □
17 ee  1 0 18 ee 1 0  6 45  6 0 30 35  6 0      □ □ □ □ □ □ □ □ □ E □ □ 0 5 □ □
 c 2f  6 0  d 44 6 0 1a ee  1 0 54 52 14 0      □ / □ □   D □ □ □ □ □ □ T R □ □
1b ee  1 0 1c ee 1 0 e2 f8  6 0 fd 2c  4 0      □ □ □ □ □ □ □ □ □ □ □ □ □ , □ □
28 c4  6 0 ee e0 5 0 1e ee  1 0 6b dc  e 0      ( □ □ □ □ □ □ □ □ □ □ □ k □ □ □
1f ee  1 0 25 ee 1 0 20 ee  1 0 22 ee  1 0      □ □ □ □ % □ □ □   □ □ □ " □ □ □
```

Il y a deux parties dans le graphe: la première est un peu chaotique, la seconde est plus régulière.

0 sur l'axe vertical du graphe signifie l'entropie la plus basse (les données qui peuvent être compressée très fortement, *ordonnées* en d'autres mots) et 8 est la plus haute (ne peuvent pas être compressées du tout, *chaotique* ou *aléatoires* en d'autres mots). Pourquoi 0 et par 8? 0 signifie 0 bits par octet (l'octet en tant que conteneur n'est pas rempli du tout) et 8 signifie 8 bits par octet, i.e., l'octet comme conteneur est complètement rempli d'information.

Donc, je mets le curseur pour pointer sur le milieu du premier bloc, et je vois clairement des tableaux d'entiers 32-bit. Maintenant je mets le curseur au milieu du second bloc et je vois un texte en anglais:



```
6c 69 73 68 69 6e 67 20 43 6f 6d 70 61 6e 79  0
43 61 6e 76 61 73 20 54 65 63 68 6e 6f 6c 6f 67
79  0 43 6f 6c 75 6d 62 75 73 20 4d 69 64 64 6c
65 20 53 63 68 6f 6f 6c  0 43 6f 61 73 74 61 6c
20 57 69 72 65 20 26 20 43 61 62 6c 65  0 43 75
72 72 65 6e 65 78  0 41 75 67 75 73 74 20 53 6f
66 74 77 61 72 65 20 43 6f 72 70 6f 72 61 74 69
6f 6e  0 41 6d 65 72 69 63 61 6e 20 41 75 74 6f
6d 6f 62 69 6c 65 20 41 73 73 6f 63 69 61 74 69
6f 6e 20 4e 61 74 6f 69 6e 61 6c 20 4f 66 66 69
63 65  0 41 63 75 72 65 78 20 45 6e 76 69 72 6f
6e 6d 65 6e 74 61 6c 20 43 6f 72 70 2e  0 50 72
69 6e 63 65 20 43 6f 72 70 6f 72 61 74 69 6f 6e
 0 47 6f 32 74 65 6c 2e 63 6f 6d  0 45 6d 70 6c
6f 79 6d 65 6e 74 20 53 65 63 75 72 69 74 79 20
43 6f 6d 6d 69 73 73 69 6f 6e  0 47 6c 6f 62 61
```

```
l i s h i n g   C o m p a n y □
C a n v a s   T e c h n o l o g
y □ C o l u m b u s   M i d d l
e   S c h o o l □ C o a s t a l
  W i r e   &   C a b l e □ C u
r r e n e x □ A u g u s t   S o
f t w a r e   C o r p o r a t i
o n □ A m e r i c a n   A u t o
m o b i l e   A s s o c i a t i
o n   N a t o i n a l   O f f i
c e □ A c u r e x   E n v i r o
n m e n t a l   C o r p . □ P r
i n c e   C o r p o r a t i o n
□ G o 2 t e l . c o m □ E m p l
o y m e n t   S e c u r i t y
C o m m i s s i o n □ G l o b a
```

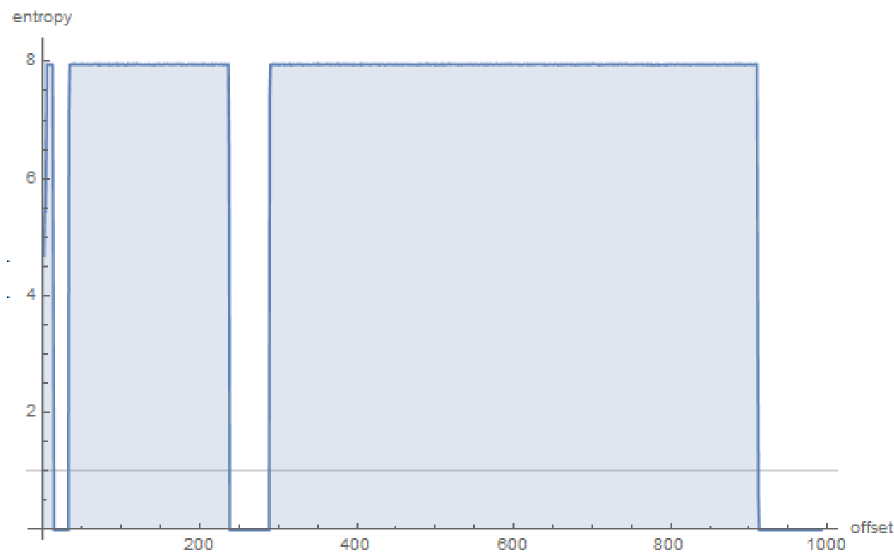
En effet, ceci sont les noms des FAIs. Donc, l'entropie de textes en anglais est 4.5-5 bits par octet? Oui, quelque chose comme ça. Wolfram Mathematica comprend quelques corpus de littérature anglaise bien connu, et nous pouvons voir l'entropie de sonnets de Shakespeare:

```
In[]:= Entropy[2,ExampleData[{"Text","ShakespearesSonnets"}]]//N
Out[]= 4.42366
```

4,4 est proche de ce que nous obtenons ()4.7-5.3). Bien sûr, les textes de la littérature anglaise classique sont quelques peu différents des noms des FAIs et autres texte en anglais que nous pouvons trouver dans des fichiers binaires (débogage/trace/messages d'erreur), mais cette valeur est proche.

Firmware TP-Link WR941

Pour l'exemple suivant, j'ai pris le firmware du routeur TP-Link WR941:



Nous voyons ici 3 blocs avec des vides. Puis le premier bloc avec une haute entropie

(démarrant à l'adresse 0) est petit, le second (adresse quelque part en 0x22000) et plus grand et le troisième (adresse 0x123000) est le plus grand. Je ne peux pas être certain de l'entropie du premier bloc, mais le 2-ème et le 3-ème ont une entropie très haute, signifiant que ces blocs sont soit compressés et/ou chiffrés.

J'ai essayé binwalk pour ce fichier de firmware:

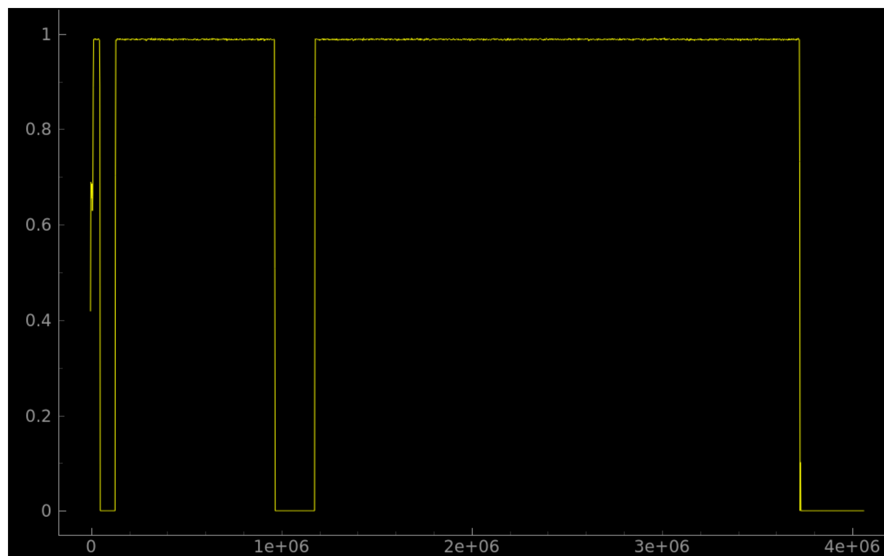
```
DECIMAL       HEXADECIMAL     DESCRIPTION
-------------------------------------------------------------------------------
0             0x0             TP-Link firmware header, firmware version: 0.-15221.3, image version: "", product ID: 0x0, product version: 155254789, kernel load address: 0x0, kernel entry point: 0x-7FFFE000, kernel offset: 4063744, kernel length: 512, rootfs offset: 837431, rootfs length: 1048576, bootloader offset: 2883584, bootloader length: 0
14832         0x39F0          U-Boot version string, "U-Boot 1.1.4 (Jun 27 2014 - 14:56:49)"
14880         0x3A20          CRC32 polynomial table, big endian
16176         0x3F30          uImage header, header size: 64 bytes, header CRC: 0x3AC66E95, created: 2014-06-27 06:56:50, image size: 34587 bytes, Data Address: 0x80010000, Entry Point: 0x80010000, data CRC: 0xDF2DBA0B, OS: Linux, CPU: MIPS, image type: Firmware Image, compression type: lzma, image name: "u-boot image"
16240         0x3F70          LZMA compressed data, properties: 0x5D, dictionary size: 33554432 bytes, uncompressed size: 90000 bytes
131584        0x20200         TP-Link firmware header, firmware version: 0.0.3, image version: "", product ID: 0x0, product version: 155254789, kernel load address: 0x0, kernel entry point: 0x-7FFFE000, kernel offset: 3932160, kernel length: 512, rootfs offset: 837431, rootfs length: 1048576, bootloader offset: 2883584, bootloader length: 0
132096        0x20400         LZMA compressed data, properties: 0x5D, dictionary size: 33554432 bytes, uncompressed size: 2388212 bytes
1180160       0x120200        Squashfs filesystem, little endian, version 4.0, compression:lzma, size: 2548511 bytes, 536 inodes, blocksize: 131072 bytes, created: 2014-06-27 07:06:52
```

En effet: il y a des choses au début, mais deux larges blocs compressés LZMA commencent en 0x20400 et 0x120200. Ce sont en gros les adresses que nous avons vu dans Mathematica. Oh, à propos, binwalk peut aussi afficher l'entropie (option -E) :

```
DECIMAL       HEXADECIMAL     ENTROPY
-------------------------------------------------------------------------------
0             0x0             Falling entropy edge (0.419187)
16384         0x4000          Rising entropy edge (0.988639)
51200         0xC800          Falling entropy edge (0.000000)
133120        0x20800         Rising entropy edge (0.987596)
968704        0xEC800         Falling entropy edge (0.508720)
1181696       0x120800        Rising entropy edge (0.989615)
3727360       0x38E000        Falling entropy edge (0.732390)
```

Les fronts ascendants correspondent à des fronts ascendants de blocs sur notre graphe. Les fronts descendants sont des points où des espaces vides commencent.

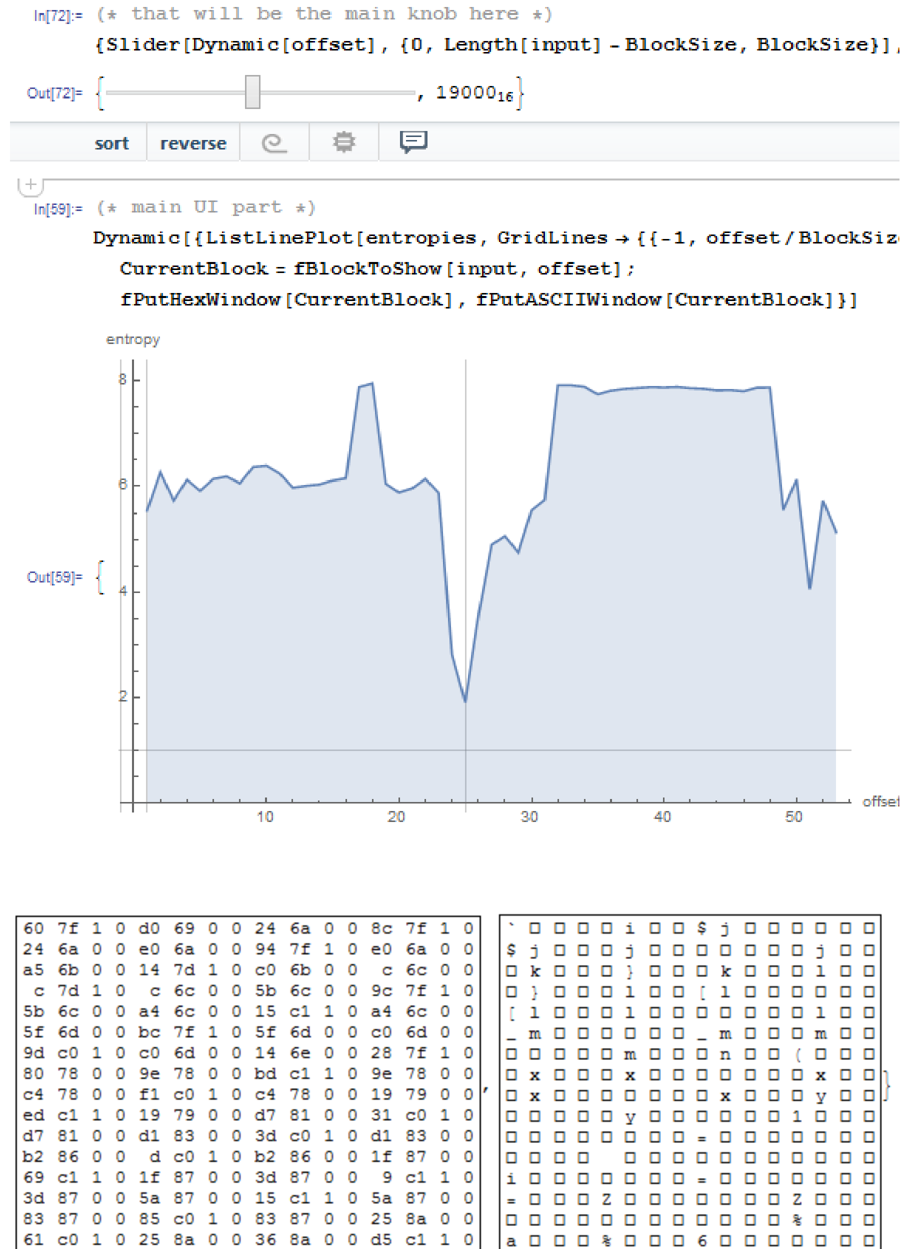
Binwalk peut aussi générer un graphe PNG (-E -J) :



Que pouvons-nous dire à propos de ces espaces vides? En regardant dans un éditeur hexadécimal, nous voyons qu'ils sont simplement remplis avec des octets à 0xFF. Pourquoi les développeurs les ont-ils mises? Peut-être parce qu'ils n'ont pas pu calculer précisément la taille des blocs compressés, et leurs ont donc alloué de l'espace avec une marge.

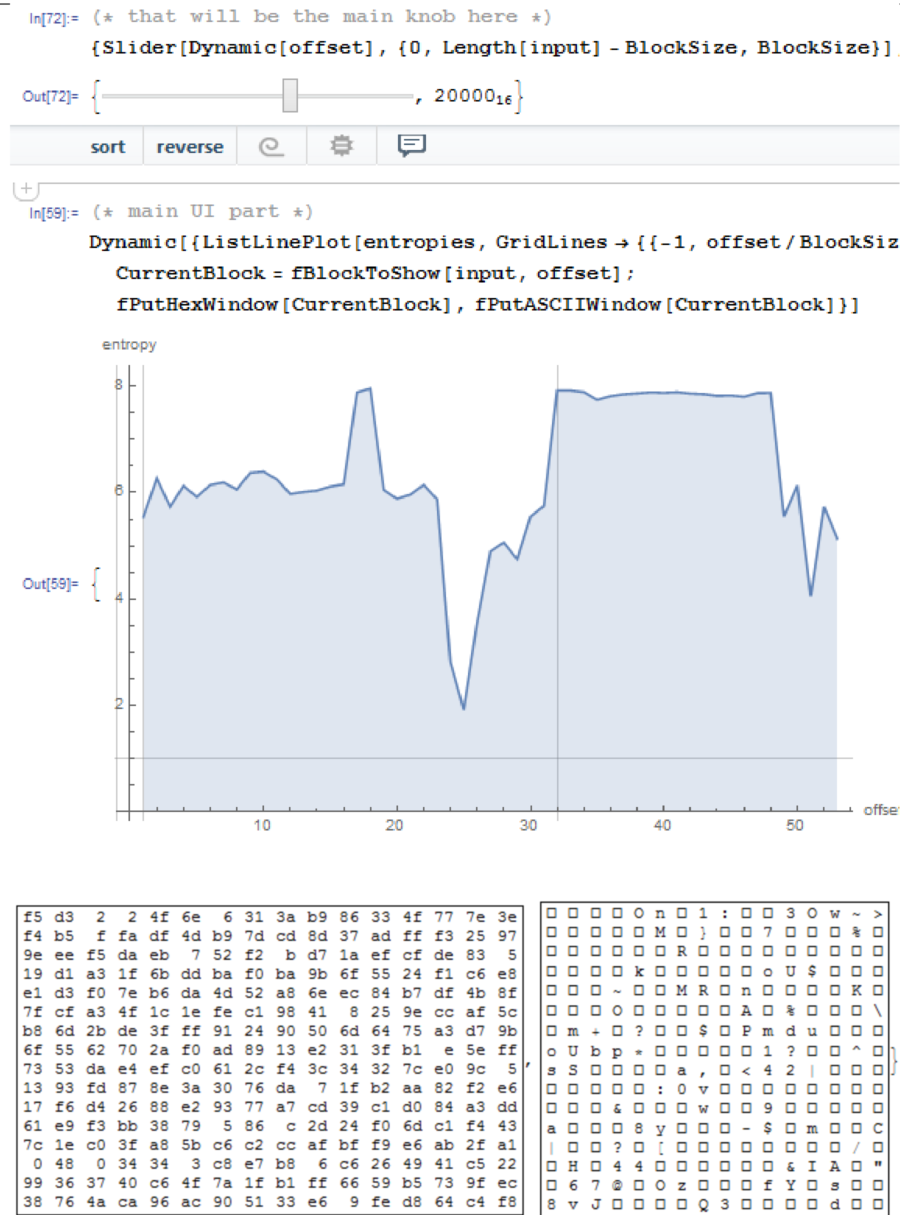
Notepad

Un autre exemple est notepad.exe que j'ai pris dans Windows 8.1:



Il y a un creux à $\approx 0x19000$ (offset absolu dans le fichier). J'ai ouvert le fichier exécutable dans un éditeur hexadécimal et trouvé des tables d'imports (qui ont une entropie plus basse que le code x86-64 dans la première moitié du graphe).

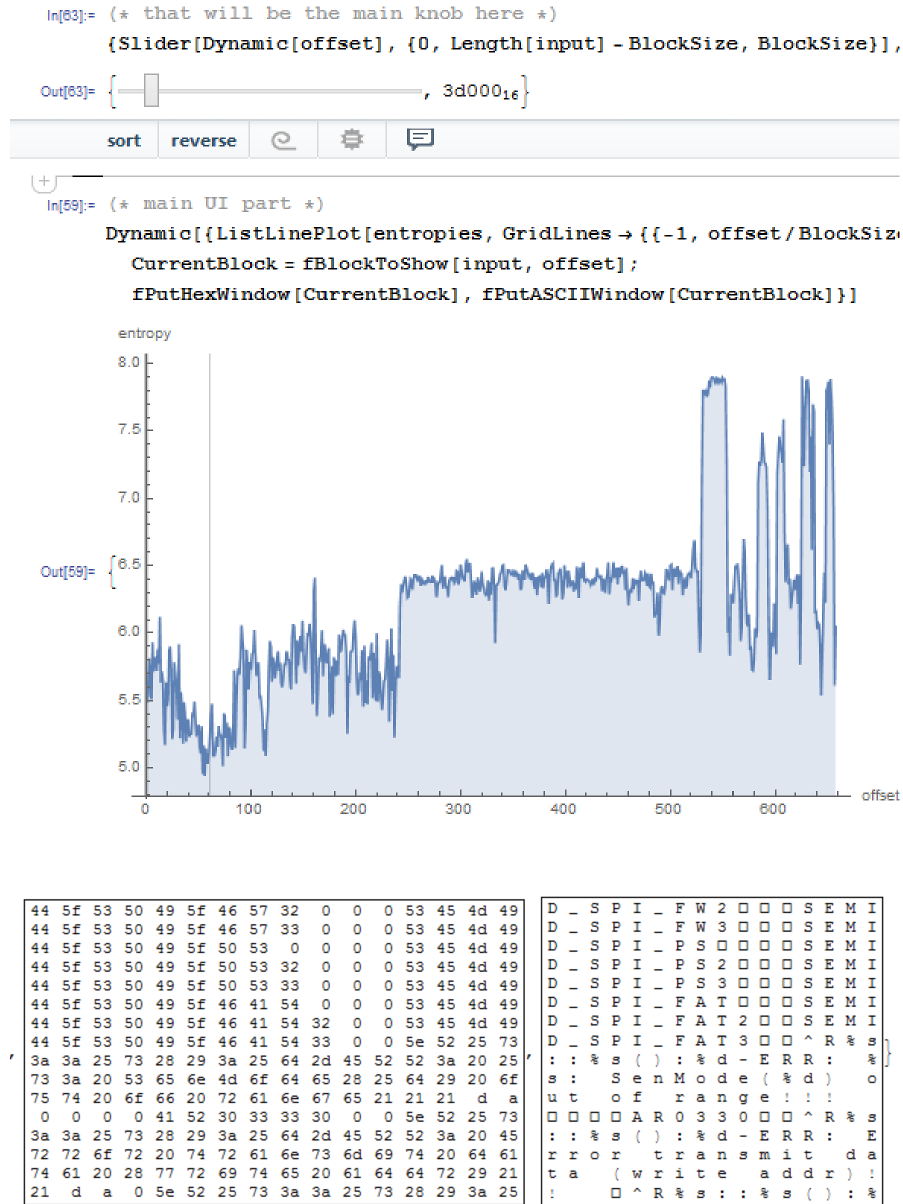
Il y a aussi un bloc avec une grande entropie qui démarre $\approx 0x20000$:



Dans un éditeur hexadécimal je peux y voir un fichier PNG, inséré dans la section ressource du fichier PE (c'est une grosse image de l'icône de notepad). Les fichiers PNG sont compressés, en effet.

Dashcam sans marque

Maintenant l'exemple le plus avancé dans cette partie est le firmware d'une dashcam sans marque que j'ai reçu d'un ami:



La creux au tout début est un texte en anglais: messages de débogage. J'ai vérifié différents ISAs et j'ai trouvé que le premier tiers du fichier complet (avec le segment de texte dedans) est en fait du code MIPS (petit-boutiste).

Par exemple, ceci est une fonction épilogue MIPS très typique:

```
ROM:000013B0                 move    $sp, $fp
ROM:000013B4                 lw      $ra, 0x1C($sp)
ROM:000013B8                 lw      $fp, 0x18($sp)
ROM:000013BC                 lw      $s1, 0x14($sp)
```

```
ROM:000013C0                 lw      $s0, 0x10($sp)
ROM:000013C4                 jr      $ra
ROM:000013C8                 addiu   $sp, 0x20
```

D'après notre graphe nous pouvons voir que le code MIPs a une entropie de 5-6 bits par octet. En effet, j'ai mesuré une fois l'entropie de différents ISAs et j'ai obtenu ces valeurs:

- x86: section .text du fichier ntoskrnl.exe de Windows 2003: 6.6
- x64: section .text du fichier ntoskrnl.exe de Windows 7 x64: 6.5
- ARM (mode thumb), Angry Birds Classic: 7.05
- ARM (mode ARM) Linux Kernel 3.8.0: 6.03
- MIPS (little endian), section .text du fichier user32.dll de Windows NT 4: 6.09

Donc l'entropie du code exécutable est plus grande que du texte en anglais, mais peut encore être compressé.

Maintenant le second tiers qui commence en 0xF5000. Je ne sais pas ce que c'est. J'ai essayé différents ISAs mais sans succès. L'entropie de ce bloc semble encore plus régulière que celui de l'exécutable. Peut-être des sortes de données?

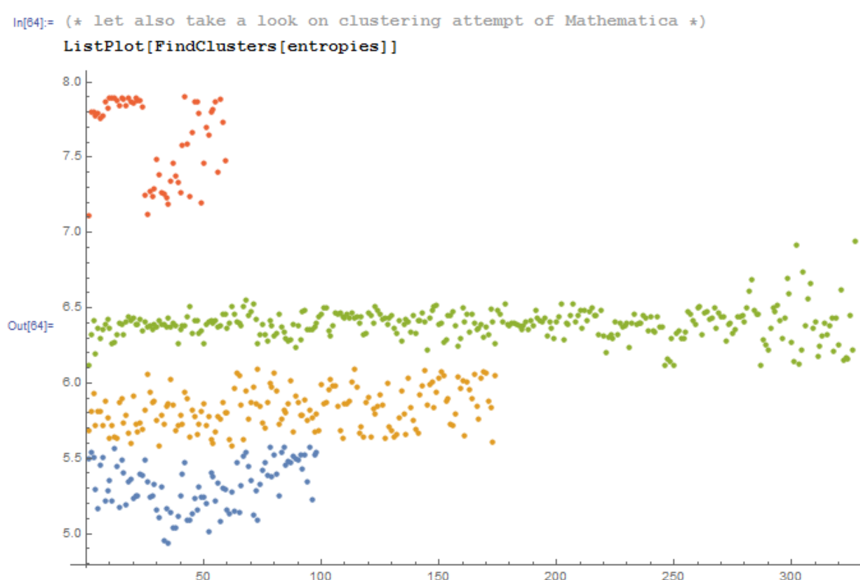
Il y a aussi un pic en $\approx 0x213000$. Je l'ai vérifié dans un éditeur hexadécimal et j'y ai trouvé un fichier JPEG (qui est, bien sûr, compressé)! Je ne sais pas ce qu'il y a à la fin. Essayons Binwalk pour ce fichier:

```
% binwalk FW96650A.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
167698        0x28F12         Unix path: /15/20/24/25/30/60/120/240fps can be served..
280286        0x446DE         Copyright string: "Copyright (c) 2012 Novatek Microelectronic Corp."
2169199       0x21196F        JPEG image data, JFIF standard 1.01
2300847       0x231BAF        MySQL MISAM compressed data file Version 3

% binwalk -E FW96650A.bin

DECIMAL       HEXADECIMAL     ENTROPY
--------------------------------------------------------------------------------
0             0x0             Falling entropy edge (0.579792)
2170880       0x212000        Rising entropy edge (0.967373)
2267136       0x229800        Falling entropy edge (0.802974)
2426880       0x250800        Falling entropy edge (0.846639)
2490368       0x260000        Falling entropy edge (0.849804)
2560000       0x271000        Rising entropy edge (0.974340)
2574336       0x274800        Rising entropy edge (0.970958)
2588672       0x278000        Falling entropy edge (0.763507)
2592768       0x279000        Rising entropy edge (0.951883)
2596864       0x27A000        Falling entropy edge (0.712814)
```

```
2600960       0x27B000       Rising entropy edge (0.968167)
2607104       0x27C800       Rising entropy edge (0.958582)
2609152       0x27D000       Falling entropy edge (0.760989)
2654208       0x288000       Rising entropy edge (0.954127)
2670592       0x28C000       Rising entropy edge (0.967883)
2676736       0x28D800       Rising entropy edge (0.975779)
2684928       0x28F800       Falling entropy edge (0.744369)
```

Oui, il trouve un fichier JPEG et même des données MySQL! Mais je ne suis pas certain que ça soit vrai—je ne l'ai pas encore vérifié.

Il est aussi intéressant d'essayer la clusterisation dans Mathematica:



Ceci est un exemple de la façon dont Mathematica groupe des valeurs d'entropie diverses dans des groupes distincts. En effet, c'est quelque chose de plausible. Les points bleus dans l'intervalle 5.0-5.5 sont probablement relatif à du texte en anglais, Les points jaunes dans 5.5-6 sont du code MIPS. Beaucoup de points verts dans 6.0-6.5 sont dans le second tiers non identifié. Les points orange proches de 8.0 sont relatifs au fichier JPEG compressé. D'autres points orange sont probablement relatif à la fin du firmware (données inconnues pour nous).

Liens

Fichiers binaires utilisés dans cette partie:
https://beginners.re/paywall/RE4B-source/current-tree//ff/entropy/files/.
Fichier notebook Wolfram Mathematica:
https://beginners.re/paywall/RE4B-source/current-tree//ff/entropy/files/binary_file_entropy.nb
(toutes les cellules doivent être évaluées pour que ça commence à fonctionner).

9.2.2 Conclusion

L'entropie peut-être utilisée comme un moyen rapide d'investigation de fichiers inconnus. En particulier, c'est un moyen rapide de trouver des morceaux de données compressées/chiffrées. Quelqu'un a dit qu'il est possible de trouver des clefs RSA[5] privées/publiques (et d'autres algorithmes cryptographiques) dans du code exécutable (les clefs ont aussi une très grande entropie), mais je n'ai pas essayé moi-même.

9.2.3 Outils

L'utilitaire Linux *ent* est très pratique pour trouver l'entropie d'un fichier[6].

Il y a un excellent visualiseur d'entropie en ligne fait par Aldo Cortesi, que j'ai essayé d'imiter avec Mathematica: `http://binvis.io`. Ses articles sur l'entropie valent la peine d'être lus: `http://corte.si/posts/visualisation/entropy/index.html`, `http://corte.si/posts/visualisation/malware/index.html`, `http://corte.si/posts/visualisation/binvis/index.html`.

Le quadriciel radare2 a la commande *#entropy* pour ceci.

Un outil pour IDA: IDAtropy[7].

9.2.4 Un mot à propos des primitives de chiffrement comme le XORage

Il est intéressant de noter que le chiffrement par un simple XOR n'affecte pas l'entropie des données. J'ai montré ceci dans l'exemple *Norton Guide* de ce livre (9.1.2 on page 1201).

Généralisation: le chiffrement par substitution n'affecte pas l'entropie des données (et XOR peut être vu comme un chiffrement par substitution). La raison est que l'algorithme de calcul de l'entropie voit les données au niveau de l'octet. D'un autre côté, les données chiffrées par un pattern XOR de 2 ou 4 octets donnent un autre niveau d'entropie.

Néanmoins, une entropie basse est en général un signe de chiffrement amateur faible (qui est aussi utilisé dans les clefs/fichiers de licence, etc.).

9.2.5 Plus sur l'entropie de code exécutable

Il est rapidement perceptible que la plus grande source d'entropie dans du code exécutable est probablement dûe aux offsets encodés dans les opcodes. Par exemple, ces deux instructions consécutives vont avoir des offsets relatifs différents dans leur opcode, alors qu'elles pointent en fait sur la même fonction:

```
function proc
...
```

[5] Rivest Shamir Adleman
[6] `http://www.fourmilab.ch/random/`
[7] `https://github.com/danigargu/IDAtropy`

```
function endp

...

CALL function
...
CALL function
```

Un compresseur de code exécutable idéal encoderait l'information comme ceci: *Il y a un CALL à "function" à l'adresse X et la même CALL à l'adresse Y* sans nécessiter d'encoder deux fois l'adresse de *function*.

Pour gérer ceci, les compresseurs de code exécutable sont parfois capable de réduire l'entropie ici. Un exemple est UPX: http://sourceforge.net/p/upx/code/ci/default/tree/doc/filter.txt.

9.2.6 PRNG

Lorsque je lance GnuPG pour générer une nouvelle clef privée (secrète), il demande de l'entropie …

```
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.

Not enough random bytes available.  Please do some other work to give
the OS a chance to collect more entropy! (Need 169 more bytes)
```

Ceci signifie qu'un bon PRNG prend longtemps pour produire des résultats avec une haute entropie, et ceci est ce dont la clef cryptographique secrète à besoin. Mais un CPRNG[8] est compliqué (car un ordinateur est lui-même un dispositif hautement déterministe), donc GnuPG demande du hasard supplémentaire à l'utilisateur.

9.2.7 Plus d'exemples

Voici un cas où j'ai essayé de calculer l'entropie de certains blocs avec du contenu inconnu: 8.9 on page 1112.

9.2.8 Entropie de fichiers variés

L'entropie de données aléatoires est proche de 8:

```
% dd bs=1M count=1 if=/dev/urandom | ent
Entropy = 7.999803 bits per byte.
```

Ceci signifie que presque tout l'espace disponible d'un octet est rempli d'information.

256 octets répartis dans l'intervalle 0..255 donnent exactement une valeur de 8:

[8]Cryptographically secure PseudoRandom Number Generator

```
#!/usr/bin/env python
import sys

for i in range(256):
    sys.stdout.write(chr(i))
```

```
% python 1.py | ent
Entropy = 8.000000 bits per byte.
```

L'ordre des octets est sans importance. Ceci signifie que tout l'espace dans un octet est rempli.

L'entropie de tout bloc rempli d'octets à zéro est 0:

```
% dd bs=1M count=1 if=/dev/zero | ent
Entropy = 0.000000 bits per byte.
```

L'entropie d'une chaîne constituée d'un seul (n'importe lequel) octet est 0:

```
% echo -n "aaaaaaaaaaaaaaaaaaa" | ent
Entropy = 0.000000 bits per byte.
```

L'entropie d'une chaîne en base64 est la même que la données source, mais multiplié par $\frac{3}{4}$. Ceci car l'encodage base64 utilise 64 symboles au lieu de 256.

```
% dd bs=1M count=1 if=/dev/urandom | base64 | ent
Entropy = 6.022068 bits per byte.
```

Peut-être que 6.02, assez proche de 6, est dû au caractère de remplissage (=) qui fausse un peu nos statistiques.

Uuencode utilise aussi 64 symboles:

```
% dd bs=1M count=1 if=/dev/urandom | uuencode - | ent
Entropy = 6.013162 bits per byte.
```

Ceci signifie que les chaînes base64 et Uuencode peuvent être transmises en utilisant des octets ou caractères sur 6-bit.

Toute information aléatoire au format hexadécimal a une entropie de 4 bits par octet:

```
% openssl rand -hex $\$$(( 2**16 )) | ent
Entropy = 4.000013 bits per byte.
```

L'entropie d'un texte en anglais pris au hasard dans la bibliothèque Gutenbert a une entropie de ≈ 4.5. La raison de ceci est que les textes anglais utilisent principalement 26 symboles, et $log_2(26) =\approx 4.7$, i.e., vous aurez besoin d'octets de 5-bit pour transmettre des textes en anglais non compressés, ça sera suffisant (ça l'était en effet au temps du télétype).

Le texte choisi au hasard dans la bibliothèque http://lib.ru est l'"Idiot"[9], de F.M.Dostoevsky qui est encodé en CP1251.

[9] http://az.lib.ru/d/dostoewskij_f_m/text_0070.shtml

Et ce fichier a une entropie de ≈ 4.98. Le russe comporte 33 caractères et $log_2(33) =\approx 5.04$. Mais il le caractère “ë” est impopulaire et rare. Et $log_2(32) = 5$ (l’alphabet russe sans ce caractère rare)—maintenant ceci est proche de ce que nous avons obtenu.

Quoiqu’il en soit, le texte dont nous parlons utilise la lettre “ë’, mais, sans doute y est-elle rarement utilisée.

Le même fichier transcodé de CP1251 en UTF-8 donne une entropie de ≈ 4.23. Chaque caractère cyrillique encodé en UTF-8 est généralement encodé en une paire, et le premier octet est toujours: 0xD0 ou 0xD1. C’est peut-être ce qui cause ce biais.

Générons des bits aléatoirement et écrivons les avec les caractères “T” et “F”:

```
#!/usr/bin/env python
import random, sys

rt=""
for i in range(102400):
    if random.randint(0,1)==1:
        rt=rt+"T"
    else:
        rt=rt+"F"
print rt
```

Échantillon: `...TTTFTFTTTFFFTTTFTTTTTTFTTFFTTTFTFTTFTTFFFFFF....`

L’entropie est très proche de 1 (i.e., 1 bit par octet).

Générons des chiffres décimaux aléatoirement:

```
#!/usr/bin/env python
import random, sys

rt=""
for i in range(102400):
    rt=rt+"%d" % random.randint(0,9)
print rt
```

Échantillon: `...52203466119390328807552582367031963888032....`

L’entropie sera proche de 3.32, en effet, c’est $log_2(10)$.

9.2.9 Réduire le niveau d’entropie

J’ai vu une fois un logiciel qui stockait chaque octet de données chiffrées sur 3 octets: chacun avait une valeur de $\approx \frac{byte}{3}$, donc reconstruire l’octet chiffré impliquait de faire la somme de 3 octets consécutifs. Ça semble absurde.

Mais certaines personnes disent que ça a été fait pour pour cacher le fait que les données contenaient quelque chose de chiffré: la mesure de l’entropie d’un tel bloc donnait une valeur bien plus faible.

9.3 Fichier de sauvegarde du jeu Millenium

« Millenium Return to Earth » est un ancien jeu DOS (1991), qui vous permet d'extraire des ressources, de construire des vaisseaux, de les équiper et de les envoyer sur d'autres planêtes, et ainsi de suite[10].

Comme beaucoup d'autres jeux, il vous permet de sauvegarder l'état du jeu dans un fichier.

Regardons si l'on peut y trouver quelque chose.

[10] Il peut être téléchargé librement ici

Donc, il y a des mines dans le jeu. Sur certaines planêtes, les mines rapportent plus vite, sur d'autres, moins vite. L'ensemble des ressources est aussi différent.

Ici, nous pouvons voir quelles ressources sont actuellement extraites.

Fig. 9.14: Mine: état 1

Sauvegardons l'état du jeu. C'est un fichier de 9538 octets.

Attendons quelques «jours» dans le jeu, et maintenant, nous avons plus de ressources extraites des mines.

Fig. 9.15: Mine: état 2

Sauvegardons à nouveau l'état du jeu.

Maintenant, essayons juste de comparer au niveau binaire les fichiers de sauvegarde en utilisant le simple utilitaire DOS/Windows FC:

```
...> FC /b 2200save.i.v1 2200SAVE.I.V2

Comparing files 2200save.i.v1 and 2200SAVE.I.V2
00000016: 0D 04
00000017: 03 04
0000001C: 1F 1E
00000146: 27 3B
00000BDA: 0E 16
00000BDC: 66 9B
00000BDE: 0E 16
00000BE0: 0E 16
00000BE6: DB 4C
00000BE7: 00 01
00000BE8: 99 E8
00000BEC: A1 F3
00000BEE: 83 C7
00000BFB: A8 28
00000BFD: 98 18
00000BFF: A8 28
```

```
00000C01: A8 28
00000C07: D8 58
00000C09: E4 A4
00000C0D: 38 B8
00000C0F: E8 68
...
```

La sortie est incomplète ici, il y a plus de différences, mais j'ai tronqué le résultat pour montrer ce qu'il y a de plus intéressant.

Dans le premier état, nous avons 14 «unités» d'hydrogène et 102 «unités» d'oxygène.

Nous avons respectivement 22 et 155 «unités» dans le second état. Si ces valeurs sont sauvées dans le fichier de sauvegarde, nous devrions les voir dans la différence. Et en effet, nous les voyons. Il y a 0x0E (14) à la position 0xBDA et cette valeur est à 0x16 (22) dans la nouvelle version du fichier. Ceci est probablement l'hydrogène. Il y a 0x66 (102) à la position 0xBDC dans la vieille version et x9B (155) dans la nouvelle version du fichier. Il semble que ça soit l'oxygène.

Les deux fichiers sont disponibles sur le site web pour ceux qui veulent les inspecter (ou expérimenter) plus: beginners.re.

Voici la nouvelle version du fichier ouverte dans Hiew, j'ai marqué les valeurs relatives aux ressources extraites dans le jeu:

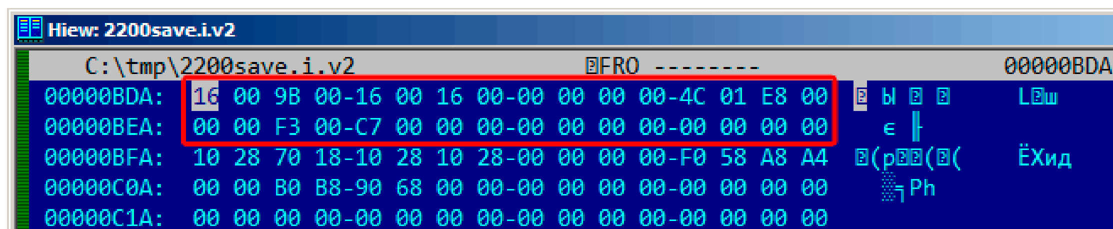


Fig. 9.16: Hiew: état 1

Vérifions chacune d'elles.

Ce sont clairement des valeurs 16-bits: ce n'est pas étonnant pour un logiciel DOS 16-bit où le type *int* fait 16-bit.

Vérifions nos hypothèses. Nous allons écrire la valeur 1234 (0x4D2) à la première position (ceci doit être l'hydrogène) :

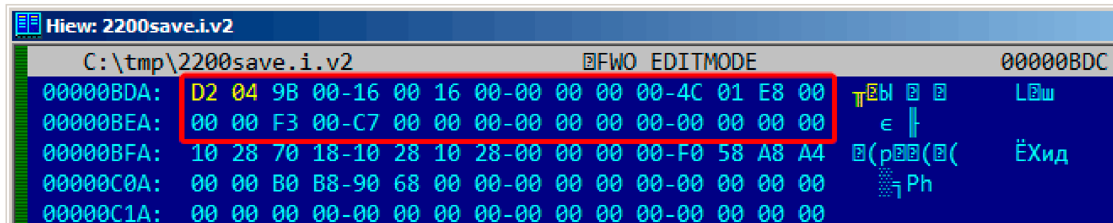


Fig. 9.17: Hiew: écrivons 1234 (0x4D2) ici

Puis nous chargeons le fichier modifié dans le jeu et jettons un coup d'œil aux statistiques des mines:

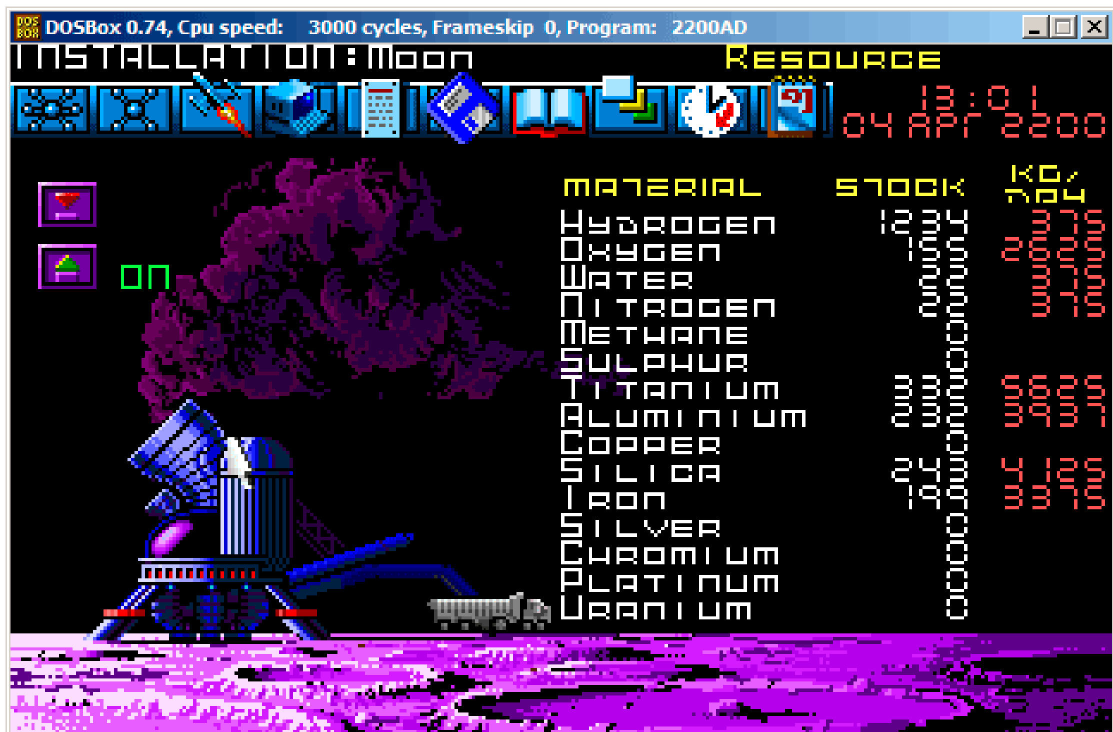


Fig. 9.18: Vérifions la valeur pour l'hydrogène

Donc oui, c'est bien ça.

Maintenant essayons de finir le jeu le plus vite possible, mettons les valeurs maximales partout:

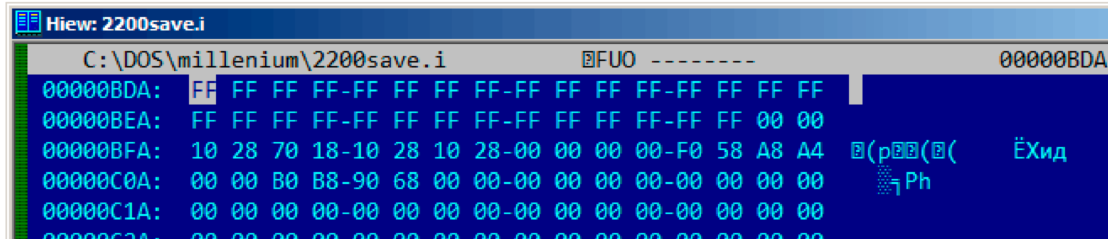


Fig. 9.19: Hiew: mettons les valeurs maximales

0xFFFF représente 65535, donc oui, nous avons maintenant beaucoup de ressources:

Fig. 9.20: Toutes les ressources sont en effet à 65535 (0xFFFF)

Laissons passer quelques «jours» dans le jeu et oups! Nous avons un niveau plus bas pour quelques ressources:

Fig. 9.21: Dépassement des variables de ressource

C'est juste un dépassement.

Le développeur du jeu n'a probablement pas pensé à un niveau aussi élevé de ressources, donc il n'a pas dû mettre des tests de dépassement, mais les mines «travaillent» dans le jeu, des ressources sont extraites, c'est pourquoi il y a des dépassements. Apparemment, c'est une mauvaise idée d'être aussi avide.

Il y a sans doute beaucoup plus de valeurs sauvées dans ce fichier.

Ceci est donc une méthode très simple de tricher dans les jeux. Les fichiers des meilleurs scores peuvent souvent être modifiés comme ceci.

Plus d'informations sur la comparaison des fichiers et des snapshots de mémoire: 5.10.2 on page 942.

9.4 *fortune* programme d'indexation de fichier

(Cette partie est tout d'abord apparue sur mon blog le 25 avril 2015.)

fortune est un programme UNIX bien connu qui affiche une phrase aléatoire depuis une collection. Certains geeks ont souvent configuré leur système de telle manière que *fortune* puisse être appelé après la connexion. *fortune* prend les phrases depuis des fichiers texte se trouvant dans */usr/share/games/fortunes* (sur Ubuntu Linux). Voici un exemple (fichier texte «fortunes») :

```
A day for firm decisions!!!!!  Or is it?
%
A few hours grace before the madness begins again.
%
A gift of a flower will soon be made to you.
%
A long-forgotten loved one will appear soon.

Buy the negatives at any price.
%
A tall, dark stranger will have more fun than you.
%
...
```

Donc, il s'agit juste de phrases, parfois sur plusieurs lignes, séparées par le signe pourcent. La tâche du programme *fortune* est de trouver une phrase aléatoire et de l'afficher. Afin de réaliser ceci, il doit scanner le fichier texte complet, compter les phrases, en choisir une aléatoirement et l'afficher. Mais le fichier texte peut être très gros, et même sur les ordinateurs modernes, cet algorithme naïf est du gaspillage de ressource. La façon simple de procéder est de garder un fichier index binaire contenant l'offset de chaque phrase du fichier texte. Avec le fichier index, le programme *fortune* peut fonctionner beaucoup plus vite: il suffit de choisir un index aléatoirement, prendre son offset, se déplacer à cet offset dans le fichier texte et d'y lire la phrase. C'est ce qui est effectivement fait dans le programme *fortune*. Examinons ce qu'il y a dans ce fichier index (ce sont les fichiers .dat dans le même répertoire) dans un éditeur hexadécimal. Ce programme est open-source bien sûr, mais intentionnellement, je ne vais pas jeter un coup d'œil dans le code source.

```
% od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 01 af 00 00 00 bb 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 2b
000020 00 00 00 60 00 00 00 8f 00 00 00 df 00 00 01 14
000030 00 00 01 48 00 00 01 7c 00 00 01 ab 00 00 01 e6
000040 00 00 02 20 00 00 02 3b 00 00 02 7a 00 00 02 c5
000050 00 00 03 04 00 00 03 3d 00 00 03 68 00 00 03 a7
000060 00 00 03 e1 00 00 04 19 00 00 04 2d 00 00 04 7f
000070 00 00 04 ad 00 00 04 d5 00 00 05 05 00 00 05 3b
000080 00 00 05 64 00 00 05 82 00 00 05 ad 00 00 05 ce
000090 00 00 05 f7 00 00 06 1c 00 00 06 61 00 00 06 7a
0000a0 00 00 06 d1 00 00 07 0a 00 00 07 53 00 00 07 9a
0000b0 00 00 07 f8 00 00 08 27 00 00 08 59 00 00 08 8b
0000c0 00 00 08 a0 00 00 08 c4 00 00 08 e1 00 00 08 f9
0000d0 00 00 09 27 00 00 09 43 00 00 09 79 00 00 09 a3
0000e0 00 00 09 e3 00 00 0a 15 00 00 0a 4d 00 00 0a 5e
0000f0 00 00 0a 8a 00 00 0a a6 00 00 0a bf 00 00 0a ef
000100 00 00 0b 18 00 00 0b 43 00 00 0b 61 00 00 0b 8e
000110 00 00 0b cf 00 00 0b fa 00 00 0c 3b 00 00 0c 66
```

```
000120 00 00 0c 85 00 00 0c b9 00 00 0c d2 00 00 0d 02
000130 00 00 0d 3b 00 00 0d 67 00 00 0d ac 00 00 0d e0
000140 00 00 0e 1e 00 00 0e 67 00 00 0e a5 00 00 0e da
000150 00 00 0e ff 00 00 0f 43 00 00 0f 8a 00 00 0f bc
000160 00 00 0f e5 00 00 10 1e 00 00 10 63 00 00 10 9d
000170 00 00 10 e3 00 00 11 10 00 00 11 46 00 00 11 6c
000180 00 00 11 99 00 00 11 cb 00 00 11 f5 00 00 12 32
000190 00 00 12 61 00 00 12 8c 00 00 12 ca 00 00 13 87
0001a0 00 00 13 c4 00 00 13 fc 00 00 14 1a 00 00 14 6f
0001b0 00 00 14 ae 00 00 14 de 00 00 15 1b 00 00 15 55
0001c0 00 00 15 a6 00 00 15 d8 00 00 16 0f 00 00 16 4e
...
```

Sans aide particulière, nous pouvons voir qu'il y a quatre éléments de 4 octets sur chaque ligne de 16 octets. Peut-être que c'est notre tableau d'index. J'essaye de charger tout le fichier comme un tableau d'entier 32-bit dans Wolfram Mathematica:

```
In[]:= BinaryReadList["c:/tmp1/fortunes.dat", "UnsignedInteger32"]

Out[]= {33554432, 2936078336, 3137339392, 251658240, 0, 37, 0, \
721420288, 1610612736, 2399141888, 3741319168, 335609856, 1208025088, \
2080440320, 2868969472, 3858825216, 537001984, 989986816, 2046951424, \
3305242624, 67305472, 1023606784, 1745027072, 2801991680, 3775070208, \
419692544, 755236864, 2130968576, 2902720512, 3573809152, 84213760, \
990183424, 1678049280, 2181365760, 2902786048, 3456434176, \
4144300032, 470155264, 1627783168, 2047213568, 3506831360, 168230912, \
1392967680, 2584150016, 4161208320, 654835712, 1493696512, \
2332557312, 2684878848, 3288858624, 3775397888, 4178051072, \
...
```

Nope, quelque chose est faux, les nombres sont étrangement gros. Mais retournons à la sortie de *od* : chaque élément de 4 octets a deux octets nuls et deux octets non nuls. Donc les offsets (au moins au début du fichier) sont au maximum 16-bit. Peut-être qu'un endianness différent est utilisé dans le fichier? L'endianness par défaut dans Mathematica est little-endian, comme utilisé dans les CPUs Intel. Maintenant, je le change en big-endian:

```
In[]:= BinaryReadList["c:/tmp1/fortunes.dat", "UnsignedInteger32",
 ByteOrdering -> 1]

Out[]= {2, 431, 187, 15, 0, 620756992, 0, 43, 96, 143, 223, 276, \
328, 380, 427, 486, 544, 571, 634, 709, 772, 829, 872, 935, 993, \
1049, 1069, 1151, 1197, 1237, 1285, 1339, 1380, 1410, 1453, 1486, \
1527, 1564, 1633, 1658, 1745, 1802, 1875, 1946, 2040, 2087, 2137, \
2187, 2208, 2244, 2273, 2297, 2343, 2371, 2425, 2467, 2531, 2581, \
2637, 2654, 2698, 2726, 2751, 2799, 2840, 2883, 2913, 2958, 3023, \
3066, 3131, 3174, 3205, 3257, 3282, 3330, 3387, 3431, 3500, 3552, \
...
```

Oui, c'est quelque chose de lisible. Je choisi un élément au hasard (3066) qui s'écrit 0xBFA en format hexadécimal. J'ouvre le fichier texte 'fortunes' dans un éditeur hexadécimal, je met l'offset 0xBFA et je vois cette phrase:

```
% od -t x1 -c --skip-bytes=0xbfa --address-radix=x fortunes
000bfa  44  6f  20  77  68  61  74  20  63  6f  6d  65  73  20  6e  61
         D   o       w   h   a   t       c   o   m   e   s       n   a
000c0a  74  75  72  61  6c  6c  79  2e  20  20  53  65  65  74  68  65
         t   u   r   a   l   l   y   .           S   e   e   t   h   e
000c1a  20  61  6e  64  20  66  75  6d  65  20  61  6e  64  20  74  68
             a   n   d       f   u   m   e       a   n   d       t   h
....
```

Ou:

```
Do what comes naturally.  Seethe and fume and throw a tantrum.
%
```

D’autres offsets peuvent aussi être essayés, oui, ils sont valides.

Je peux aussi vérifier dans Mathematica que chaque élément consécutif est plus grand que le précédent. I.e., les éléments du tableau sont croissants. Dans le jargon mathématiques, ceci est appelé *fonction monotone strictement croissante*.

```
In[]:= Differences[input]

Out[]= {429, -244, -172, -15, 620756992, -620756992, 43, 53, 47, \
80, 53, 52, 52, 47, 59, 58, 27, 63, 75, 63, 57, 43, 63, 58, 56, 20, \
82, 46, 40, 48, 54, 41, 30, 43, 33, 41, 37, 69, 25, 87, 57, 73, 71, \
94, 47, 50, 50, 21, 36, 29, 24, 46, 28, 54, 42, 64, 50, 56, 17, 44, \
28, 25, 48, 41, 43, 30, 45, 65, 43, 65, 43, 31, 52, 25, 48, 57, 44, \
69, 52, 62, 73, 62, 53, 37, 68, 71, 50, 41, 57, 69, 58, 70, 45, 54, \
38, 45, 50, 42, 61, 47, 43, 62, 189, 61, 56, 30, 85, 63, 48, 61, 58, \
81, 50, 55, 63, 83, 80, 49, 42, 94, 54, 67, 81, 52, 57, 68, 43, 28, \
120, 64, 53, 81, 33, 82, 88, 29, 61, 32, 75, 63, 70, 47, 101, 60, 79, \
33, 48, 65, 35, 59, 47, 55, 22, 43, 35, 102, 53, 80, 65, 45, 31, 29, \
69, 32, 25, 38, 34, 35, 49, 59, 39, 41, 18, 43, 41, 83, 37, 31, 34, \
59, 72, 72, 81, 77, 53, 53, 50, 51, 45, 53, 39, 70, 54, 103, 33, 70, \
51, 95, 67, 54, 55, 65, 61, 54, 54, 53, 45, 100, 63, 48, 65, 71, 23, \
28, 43, 51, 61, 101, 65, 39, 78, 66, 43, 36, 56, 40, 67, 92, 65, 61, \
31, 45, 52, 94, 82, 82, 91, 46, 76, 55, 19, 58, 68, 41, 75, 30, 67, \
92, 54, 52, 108, 60, 56, 76, 41, 79, 54, 65, 74, 112, 76, 47, 53, 61, \
66, 53, 28, 41, 81, 75, 69, 89, 63, 60, 18, 18, 50, 79, 92, 37, 63, \
88, 52, 81, 60, 80, 26, 46, 80, 64, 78, 70, 75, 46, 91, 22, 63, 46, \
34, 81, 75, 59, 62, 66, 74, 76, 111, 55, 73, 40, 61, 55, 38, 56, 47, \
78, 81, 62, 37, 41, 60, 68, 40, 33, 54, 34, 41, 36, 49, 44, 68, 51, \
50, 52, 36, 53, 66, 46, 41, 45, 51, 44, 44, 33, 72, 40, 71, 57, 55, \
39, 66, 40, 56, 68, 43, 88, 78, 30, 54, 64, 36, 55, 35, 88, 45, 56, \
76, 61, 66, 29, 76, 53, 96, 36, 46, 54, 28, 51, 82, 53, 60, 77, 21, \
84, 53, 43, 104, 85, 50, 47, 39, 66, 78, 81, 94, 70, 49, 67, 61, 37, \
51, 91, 99, 58, 51, 49, 46, 68, 72, 40, 56, 63, 65, 41, 62, 47, 41, \
43, 30, 43, 67, 78, 80, 101, 61, 73, 70, 41, 82, 69, 45, 65, 38, 41, \
57, 82, 66}
```

Comme on le voit, excepté les 6 premières valeurs (qui appartiennent sans doute à l’entête du fichier d’index), tous les nombres sont en fait la longueur des phrases

de texte (l'offset de la phrase suivante moins l'offset de la phrase courante est la longueur de la phrase courante).

Il est très important de garder à l'esprit que l'endianness peut être confondu avec un début de tableau incorrect. En effet, dans la sortie d'*od* nous voyons que chaque élément débutait par deux zéros. Mais lorsque nous décalons les des octets de chaque côté, nous pouvons interpréter ce tableau comme little-endian:

```
% od -t x1 --address-radix=x --skip-bytes=0x32 fortunes.dat
000032 01 48 00 00 01 7c 00 00 01 ab 00 00 01 e6 00 00
000042 02 20 00 00 02 3b 00 00 02 7a 00 00 02 c5 00 00
000052 03 04 00 00 03 3d 00 00 03 68 00 00 03 a7 00 00
000062 03 e1 00 00 04 19 00 00 04 2d 00 00 04 7f 00 00
000072 04 ad 00 00 04 d5 00 00 05 05 00 00 05 3b 00 00
000082 05 64 00 00 05 82 00 00 05 ad 00 00 05 ce 00 00
000092 05 f7 00 00 06 1c 00 00 06 61 00 00 06 7a 00 00
0000a2 06 d1 00 00 07 0a 00 00 07 53 00 00 07 9a 00 00
0000b2 07 f8 00 00 08 27 00 00 08 59 00 00 08 8b 00 00
0000c2 08 a0 00 00 08 c4 00 00 08 e1 00 00 08 f9 00 00
0000d2 09 27 00 00 09 43 00 00 09 79 00 00 09 a3 00 00
0000e2 09 e3 00 00 0a 15 00 00 0a 4d 00 00 0a 5e 00 00
...
```

Si nous interprétons ce tableau en little-endian, le premier élément est 0x4801, le second est 0x7C01, etc. La partie 8-bit haute de chacune de ces valeurs 16-bit nous semble être aléatoire, et la partie 8-bit basse semble être ascendante.

Mais je suis sûr que c'est un tableau en big-endian, car le tout dernier élément 32-bit du fichier est big-endian. (*00 00 5f c4* ici) :

```
% od -t x1 --address-radix=x fortunes.dat
...
000660 00 00 59 0d 00 00 59 55 00 00 59 7d 00 00 59 b5
000670 00 00 59 f4 00 00 5a 35 00 00 5a 5e 00 00 5a 9c
000680 00 00 5a cb 00 00 5a f4 00 00 5b 1f 00 00 5b 3d
000690 00 00 5b 68 00 00 5b ab 00 00 5b f9 00 00 5c 49
0006a0 00 00 5c ae 00 00 5c eb 00 00 5d 34 00 00 5d 7a
0006b0 00 00 5d a3 00 00 5d f5 00 00 5e 3a 00 00 5e 67
0006c0 00 00 5e a8 00 00 5e ce 00 00 5e f7 00 00 5f 30
0006d0 00 00 5f 82 00 00 5f c4
0006d8
```

Peut-être que le développeur du programme *fortune* avait un ordinateur big-endian ou peut-être a-t-il été porté depuis quelque chose comme ça.

Ok, donc le tableau est big-endian, et, à en juger avec bon sens, la toute première phrase dans le fichier texte doit commencer à l'offset zéro. Donc la valeur zéro doit se trouver dans le tableau quelque part au tout début. Nous avons un couple d'élément à zéro au début. Mais le second est plus tentant: 43 se trouve juste après et 43 est un offset valide sur une phrase anglaise correcte dans le fichier texte.

Le dernier élément du tableau est 0x5FC4, et il n'y a pas de tel octet à cet offset dans le fichier texte. Donc le dernier élément du tableau pointe au delà de la fin du fichier. C'est probablement ainsi car la longueur de la phrase est calculée comme la

différence entre l'offset de la phrase courante et l'offset de la phrase suivante. Ceci est plus rapide que de lire la chaîne à la recherche du caractère pourcent. Mais ceci ne fonctionne pas pour le dernier élément. Donc un élément *fictif* est aussi ajouté à la fin du tableau.

Donc les 6 première valeur entière 32-bit sont une sorte d'en-tête.

Oh, j'ai oublié de compter les phrases dans le fichier texte:

```
% cat fortunes | grep % | wc -l
432
```

Le nombre de phrases peut être présent dans dans l'index, mais peut-être pas. Dans le cas de fichiers d'index simples, le nombre d'éléments peut être facilement déduit de la taille du fichier d'index. Quoiqu'il en soit, il y a 432 phrases dans le fichier texte. Et nous voyons quelque chose de très familier dans le second élément (la valeur 431). J'ai vérifié dans d'autres fichiers (literature.dat et riddles.dat sur Ubuntu Linux) et oui, le second élément 32-bit est bien le nombre de phrases moins 1. Pourquoi *moins 1* ? Peut-être que ceci n'est pas le nombre de phrases, mais plutôt le numéro de la dernière phrase (commençant à zéro)?

Et il y a d'autres éléments dans l'entête. Dans Mathematica, je charge chacun des trois fichiers disponible et je regarde leur en-tête:

```
In[14]:= input = BinaryReadList["c:/tmp1/fortunes.dat", "UnsignedInteger32",
           ByteOrdering → 1];

In[18]:= BaseForm[Take[input, {1, 6}], 16]
Out[18]//BaseForm=
         {2₁₆, 1af₁₆, bb₁₆, f₁₆, 0₁₆, 25000000₁₆}

In[19]:= input = BinaryReadList["c:/tmp1/literature.dat", "UnsignedInteger32",
           ByteOrdering → 1];

In[20]:= BaseForm[Take[input, {1, 6}], 16]
Out[20]//BaseForm=
         {2₁₆, 106₁₆, 983₁₆, 1a₁₆, 0₁₆, 25000000₁₆}

In[21]:= input = BinaryReadList["c:/tmp1/riddles.dat", "UnsignedInteger32", ByteOrdering → 1];

In[22]:= BaseForm[Take[input, {1, 6}], 16]
Out[22]//BaseForm=
         {2₁₆, 80₁₆, 7f2₁₆, 24₁₆, 0₁₆, 25000000₁₆}
```

Je n'ai aucune idée de la signification des autres valeurs, excepté la taille du fichier d'index. Certains champs sont les même pour tous les fichiers, d'autres non. D'après mon expérience, ils peuvent être:

- signature de fichier;
- version de fichier;
- checksum;

- des flags;
- peut-être même des identifiants de langages;
- timestamp du fichier texte, donc le programme *fortune* regénèrerait le fichier d'index isi l'utilisateur modifiait le fichier texte.

Par exemple, les fichiers Oracle .SYM (9.5 on the following page) qui contiennent la table des symboles pour les fichiers DLL, contiennent aussi un timestamp correspondant au fichier DLL, afin d'être sûr qu'il est toujours valide.

D'un autre côté, les timestamps des fichiers textes et des fichiers d'index peuvent être désynchronisés après archivage/désarchivage/installation/déploiement/etc.

Mais ce ne sont pas des timestamps, d'après moi. La manière la plus compacte de représenter la date et l'heure est la valeur UNIX, qui est un nombre 32-bit. Nous ne voyons rien de tel ici. D'autres façons de les représenter sont encore moins compactes.

Donc, voici supposément comment fonctionne l'algorithme de *fortune* :

- prendre le nombre du second élément de la dernière phrase;
- générer un nombre aléatoire dans l'intervalle 0..number_of_last_phrase;
- trouver l'élément correspondant dans le tableau des offsets, prendre aussi l'offset suivant;
- écrire sur *stdout* tous les caractères depuis le fichier texte en commençant à l'offset jusqu'à l'offset suivant moins 2 (afin d'ignorer le caractère pourcent terminal et le caractère de la phrase suivante).

9.4.1 Hacking

Effectuons quelques essais afin de vérifier nos hypothèses. Je vais créer ce fichier texte dans le chemin et le nom */usr/share/games/fortunes/fortunes* :

```
Phrase one.
%
Phrase two.
%
```

Puis, ce fichier fortunes.dat. Je prend l'entête du fichier original fortunes.dat, j'ai mis à zéro changé le second champ (nombre de phrases) et j'ai laissé deux éléments dans le tableau: 0 et 0x1c, car la longueur totale du fichier texte *fortunes* est 28 (0x1c) octets:

```
% od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 00 00 00 00 bb 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 1c
```

Maintenant, je le lance:

```
% /usr/games/fortune
fortune: no fortune found
```

Quelque chose ne va pas. Mettons le second champ à 1:

```
% od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 01 00 00 00 bb 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 1c
```

Maintenant, ça fonctionne. Il affiche seulement la première phrase:

```
% /usr/games/fortune
Phrase one.
```

Hmmm. Laissons seulement un élément dans le tableau (0) sans le terminer:

```
% od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 01 00 00 00 bb 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00
00001c
```

Le programme Fortune montre toujours seulement la première phrase.

De cet essai nous apprenons que la signe pourcent dans le fichier texte est analysé et la taille n'est pas calculée comme je l'avais déduit avant, peut-être que même l'élément terminal du tableau n'est pas utilisé. Toutefois, il peut toujours l'être. Et peut-être qu'il l'a été dans le passé?

9.4.2 Les fichiers

Pour les besoins de la démonstration, je ne regarde toujours pas dans le code source de *fortune*. Si vous soulez essayer de comprendre la signification des autres valeurs dans l'entête du fichier d'index, vous pouvez essayer de le faire sans regarder dans le code source. Les fichiers que j'ai utilisé sous Ubuntu Linux 14.04 sont ici: http://beginners.re/examples/fortune/, les fichiers bricolés le sont aussi.

Oh, j'ai pris les fichiers de la version x64 d'Ubuntu, mais les éléments du tableau ont toujours une taille de 32 bit. C'est parce que les fichiers de texte de *fortune* ne vont sans doute jamais dépasser une taille de 4GiB[11]. Mais s'ils le devaient, tous les éléments devraient avoir une taille de 64 bit afin de pouvoir stocker un offset de dans un fichier texte plus gros que 4GiB.

Pour les lecteurs impatients, le code source de *fortune* est ici: https://launchpad.net/ubuntu/+source/fortune-mod/1:1.99.1-3.1ubuntu4.

9.5 Oracle RDBMS : fichiers .SYM

Lorsqu'un processus d'Oracle RDBMS rencontre un sorte de crash, il écrit beaucoup d'information dans les fichiers de trace, incluant la trace de la pile, comme ceci:

```
----- Call Stack Trace -----
calling              call     entry                argument values in hex
```

[11] Gibibyte

```
location             type     point                (? means dubious value)
-------------------- -------- -------------------- ⤦
    ↳ ----------------------------
_kqvrow()                     00000000
_opifch2()+2729      CALLptr  00000000             23D4B914 E47F264 1F19AE2
                                                   EB1C8A8 1
_kpoal8()+2832       CALLrel  _opifch2()           89 5 EB1CC74
_opiodr()+1248       CALLreg  00000000             5E 1C EB1F0A0
_ttcpip()+1051       CALLreg  00000000             5E 1C EB1F0A0 0
_opitsk()+1404       CALL???  00000000             C96C040 5E EB1F0A0 0 ⤦
    ↳ EB1ED30
                                                   EB1F1CC 53E52E 0 EB1F1F8
_opiino()+980        CALLrel  _opitsk()            0 0
_opiodr()+1248       CALLreg  00000000             3C 4 EB1FBF4
_opidrv()+1201       CALLrel  _opiodr()            3C 4 EB1FBF4 0
_sou2o()+55          CALLrel  _opidrv()            3C 4 EB1FBF4
_opimai_real()+124   CALLrel  _sou2o()             EB1FC04 3C 4 EB1FBF4
_opimai()+125        CALLrel  _opimai_real()       2 EB1FC2C
_OracleThreadStart@  CALLrel  _opimai()            2 EB1FF6C 7C88A7F4 ⤦
    ↳ EB1FC34 0
4()+830                                            EB1FD04
77E6481C             CALLreg  00000000             E41FF9C 0 0 E41FF9C 0 ⤦
    ↳ EB1FFC4
00000000             CALL???  00000000
```

Mais bien sûr, les exécutables d'Oracle RDBMS doivent avoir une sorte d'information de débogage ou de fichiers de carte avec l'information sur les symboles incluse ou quelque chose comme ça.

Oracle RDBMS sur Windows NT a l'information sur les symboles incluse dans des fichiers avec l'extension .SYM, mais le format est propriétaire. (Les fichiers texte en clair sont bons, mais nécessite une analyse supplémentaire, d'où un accès plus lent.)

Voyons si nous pouvons comprendre son format.

Nous allons choisir le plus petit fichier `orawtc8.sym` qui vient avec le fichier `orawtc8.dll` dans Oracle 8.1.7[12].

[12] Nous pouvons choisir une version plus ancienne d'Oracle RDBMS intentionnellement à cause de la plus petite taille de ses modules.

Voici le fichier ouvert dans Hiew:

```
Hiew: orawtc8.sym
   C:\P\oracle_sym\v9\orawtc8.sym      FRO --------        00000000 Hiew 8.02 (c)SEN
OSYMB
_ax_reg _ax_unreg _loaddll _wtcsrin0 _wtcsrin _wtcsrfre _w
tclkm _wtcstu _wtcMerr _wtcLerr _wtchtalo _wtchtfre _slzgetevar _LhtStrCreate _LhtStrSearch _LhtStr
Insert _kpuhhalo _kpugdr _kpuhhfre _GetProcAddress@8 _LoadLibraryA@4 __CRT_INIT@12 __DllMainCRTStar
tup@12 _strncpy _free __initterm __onexit _atexit _malloc _DllMain@12 ___dllonexit _DisableThreadLi
braryCalls@4 __imp__LoadLibraryA@4 __imp__DisableThreadLibraryCalls@4 __imp__GetProcAddress@8 \177K
ERNEL32_NULL_THUNK_DATA __imp___onexit __imp__malloc __imp__free __imp___adjust_fdiv __imp__strncpy
 __imp____dllonexit __imp___initterm \177MSVCRT_NULL_THUNK_DATA __imp__kpugdr __imp__kpuhhfre __imp
__kpuhhalo \177oraclient8_NULL_THUNK_DATA __imp__LhtStrInsert __imp__slzgetevar __imp__LhtStrSearch
 __imp__LhtStrCreate \177oracore8_NULL_THUNK_DATA __IMPORT_DESCRIPTOR_oracore8 __IMPORT_DESCRIPTOR_
oraclient8 __IMPORT_DESCRIPTOR_KERNEL32 __IMPORT_DESCRIPTOR_MSVCRT __NULL_IMPORT_DESCRIPTOR ___xc_a
 ___xc_z ___xi_a ___xi_z __adjust_fdiv ___onexitend ___onexitbegin __pRawDllMain OSYM

1  2  3  4ReLoad 5  6String 7Direct 8Table 9  10Leave 11  12
```

Fig. 9.22: Le fichier entier dans Hiew

En comparant le fichier avec d'autres fichiers .SYM, nous voyons rapidement que OSYM est toujours en entête (et en fin de fichier), donc c'est peut-être la signature du fichier.

Nous voyons que, en gros, le format de fichier est: OSYM + des données binaires + un zéro délimiteur de chaîne de texte + OSYM. Les chaînes sont évidemment les noms des fonctions et des variables globales.

Nous allons marquer les signatures OSYM et les chaînes ici:

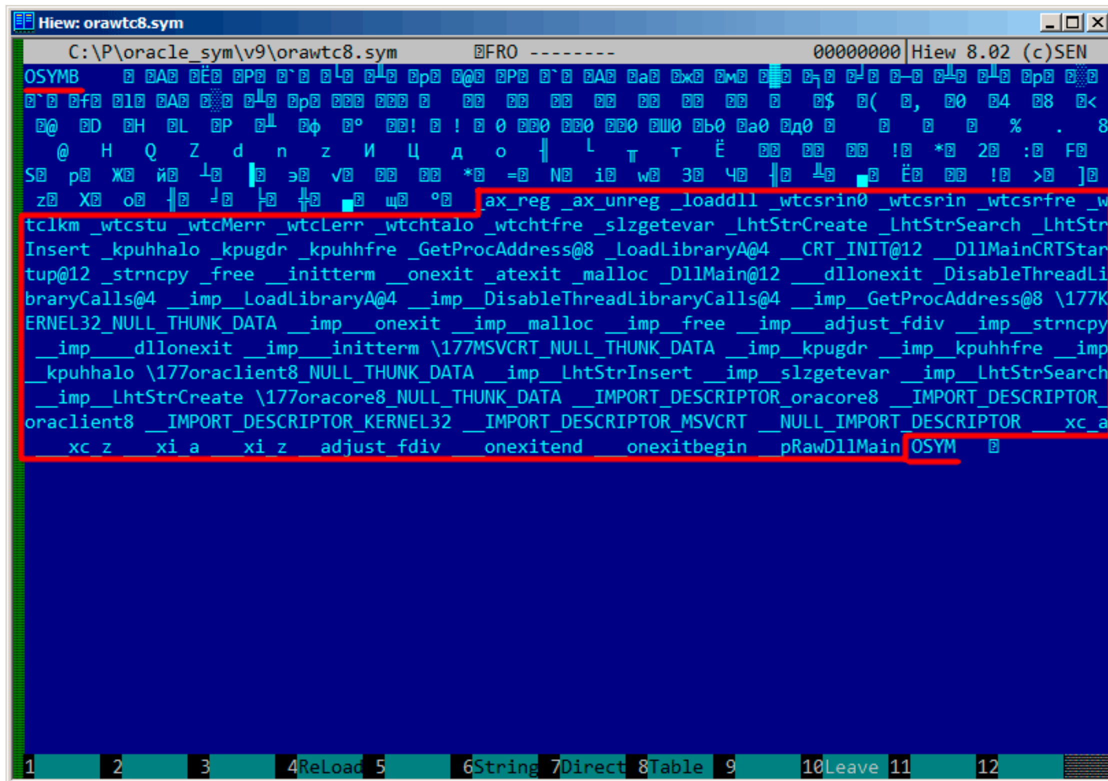


Fig. 9.23: Signatures OSYM et chaînes de texte

Bon, voyons. Dans Hiew, nous allons marquer le bloc de chaînes complet (excepté les signatures OSYM) et les mettre dans un fichier séparé. Puis, nous lançons les utilitaires UNIX *strings* et *wc* pour compter les chaînes de texte:

```
strings strings_block | wc -l
66
```

Donc, il y a 66 chaînes de texte. Veuillez noter ce nombre.

Nous pouvons dire, en général, comme une règle, que le nombre de *quelque chose* est souvent stocké séparément dans des fichiers binaires.

C'est en effet ainsi, nous pouvons trouver la valeur 66 (0x42) au début du fichier, juste après la signature OSYM:

```
$ hexdump -C orawtc8.sym
00000000  4f 53 59 4d 42 00 00 00  00 10 00 10 80 10 00 10  |OSYMB↙
    ↳ ...........|
00000010  f0 10 00 10 50 11 00 10  60 11 00 10 c0 11 00 10  |....P↙
    ↳ ...`.......|
```

```
00000020  d0 11 00 10 70 13 00 10  40 15 00 10 50 15 00 10  |....p...@...P↙
    ↪ ...|
00000030  60 15 00 10 80 15 00 10  a0 15 00 10 a6 15 00 10  ↙
    ↪ |`...............|
....
```

Bien sûr, 0x42 n'est pas ici un octet, mais plus probablement une velaur 32-bit packée en petit-boutiste, ainsi nous pouvons voir 0x42 et ensuite au moins 3 octets à zéro.

Pourquoi croyons-nous que c'est 32-bit? Car les fichiers de symboles d'Oracle RDBMS peuvent être plutôt gros.

Le fichier oracle.sym pour l'exécutable principal oracle.exe (version 10.2.0.4) contient `0x3A38E` (238478) symboles.

Nous pouvons vérifier d'autres fichiers .SYM comme ceci et ça prouve notre supposition: la valeur après la signature 32-bit OSYM représente toujours le nombre de chaînes de texte dans le fichier.

C'est une caractéristique générale de presque tous les fichiers binaires: un entête avec une signature ainsi que d'autres informations sur le fichier.

Maintenant, examinons de plus près ce qu'est ce bloc binaire.

En utilisant Hiew, nous extrayons le bloc débutant à l'offset 8 (i.e., après la valeur 32-bit *count*) se terminant au bloc de chaînes, dans un fichier binaire séparé.

Voyons le bloc binaire dans Hiew:

```
Hiew: asd2
   C:\P\oracle_sym\v9\asd2                    FRO --------                    00000000
00000000:  00 10 00 10-80 10 00 10-F0 10 00 10-50 11 00 10
00000010:  60 11 00 10-C0 11 00 10-D0 11 00 10-70 13 00 10
00000020:  40 15 00 10-50 15 00 10-60 15 00 10-80 15 00 10
00000030:  A0 15 00 10-A6 15 00 10-AC 15 00 10-B2 15 00 10
00000040:  B8 15 00 10-BE 15 00 10-C4 15 00 10-CA 15 00 10
00000050:  D0 15 00 10-E0 15 00 10-B0 16 00 10-60 17 00 10
00000060:  66 17 00 10-6C 17 00 10-80 17 00 10-B0 17 00 10
00000070:  D0 17 00 10-E0 17 00 10-10 18 00 10-16 18 00 10
00000080:  00 20 00 10-04 20 00 10-08 20 00 10-0C 20 00 10
00000090:  10 20 00 10-14 20 00 10-18 20 00 10-1C 20 00 10
000000A0:  20 20 00 10-24 20 00 10-28 20 00 10-2C 20 00 10
000000B0:  30 20 00 10-34 20 00 10-38 20 00 10-3C 20 00 10
000000C0:  40 20 00 10-44 20 00 10-48 20 00 10-4C 20 00 10
000000D0:  50 20 00 10-D0 20 00 10-E4 20 00 10-F8 20 00 10
000000E0:  0C 21 00 10-20 21 00 10-00 30 00 10-04 30 00 10
000000F0:  08 30 00 10-0C 30 00 10-98 30 00 10-9C 30 00 10
00000100:  A0 30 00 10-A4 30 00 10-00 00 00 00-08 00 00 00
00000110:  12 00 00 00-1B 00 00 00-25 00 00 00-2E 00 00 00
00000120:  38 00 00 00-40 00 00 00-48 00 00 00-51 00 00 00
00000130:  5A 00 00 00-64 00 00 00-6E 00 00 00-7A 00 00 00
00000140:  88 00 00 00-96 00 00 00-A4 00 00 00-AE 00 00 00
00000150:  B6 00 00 00-C0 00 00 00-D2 00 00 00-E2 00 00 00
00000160:  F0 00 00 00-07 01 00 00-10 01 00 00-16 01 00 00
00000170:  21 01 00 00-2A 01 00 00-32 01 00 00-3A 01 00 00
00000180:  46 01 00 00-53 01 00 00-70 01 00 00-86 01 00 00
00000190:  A9 01 00 00-C1 01 00 00-DE 01 00 00-ED 01 00 00
000001A0:  FB 01 00 00-07 02 00 00-1B 02 00 00-2A 02 00 00
1Global 2FilBlk 3CryBlk 4ReLoad 5 6String 7Direct 8Table 9 10Leave 11
```

Fig. 9.24: Bloc binaire

Il y un motif clair dedans.

Nous ajoutons des lignes rouges pour diviser le bloc:

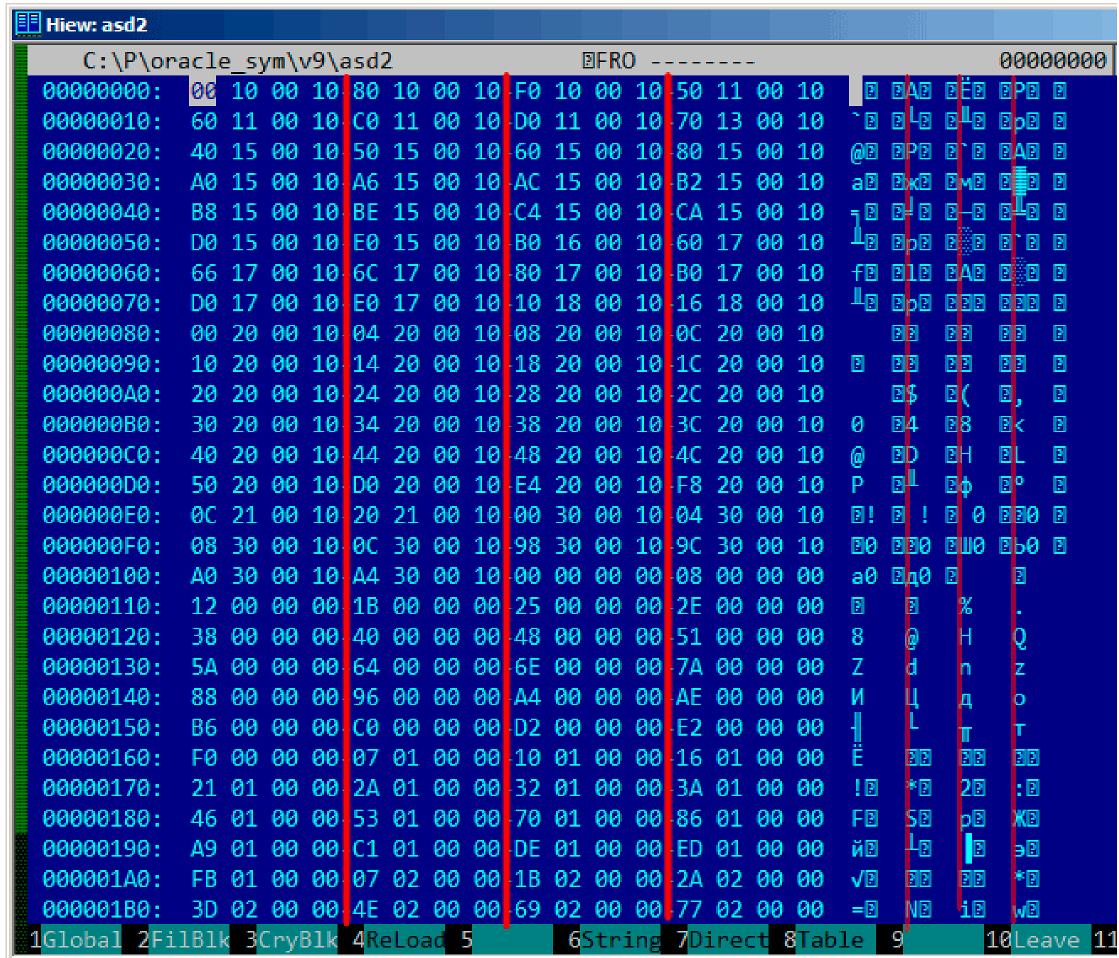


Fig. 9.25: Schéma de bloc binaire

Hiew, comme presque n'importe quel autre éditeur hexadécimal, montre 16 octets par ligne. Donc le motif est clairement visible: il y a 4 valeurs 32-bit par ligne.

Le schéma est visible visuellement car certaines valeurs ici (jusqu'à l'adresse 0x104 sont toujours de la forme 0x1000xxxx, commençant par 0x10 et des octets à zéro.

D'autres valeurs (commençant à 0x108) sont de la forme 0x0000xxxx, donc commencent toujours par deux octets à zéro.

Affichons le bloc comme un tableau de valeurs 32-bit:

Listing 9.10 : la première colonne est l'adresse

```
$ od -v -t x4 binary_block
0000000 10001000 10001080 100010f0 10001150
```

```
0000020 10001160 100011c0 100011d0 10001370
0000040 10001540 10001550 10001560 10001580
0000060 100015a0 100015a6 100015ac 100015b2
0000100 100015b8 100015be 100015c4 100015ca
0000120 100015d0 100015e0 100016b0 10001760
0000140 10001766 1000176c 10001780 100017b0
0000160 100017d0 100017e0 10001810 10001816
0000200 10002000 10002004 10002008 1000200c
0000220 10002010 10002014 10002018 1000201c
0000240 10002020 10002024 10002028 1000202c
0000260 10002030 10002034 10002038 1000203c
0000300 10002040 10002044 10002048 1000204c
0000320 10002050 100020d0 100020e4 100020f8
0000340 1000210c 10002120 10003000 10003004
0000360 10003008 1000300c 10003098 1000309c
0000400 100030a0 100030a4 00000000 00000008
0000420 00000012 0000001b 00000025 0000002e
0000440 00000038 00000040 00000048 00000051
0000460 0000005a 00000064 0000006e 0000007a
0000500 00000088 00000096 000000a4 000000ae
0000520 000000b6 000000c0 000000d2 000000e2
0000540 000000f0 00000107 00000110 00000116
0000560 00000121 0000012a 00000132 0000013a
0000600 00000146 00000153 00000170 00000186
0000620 000001a9 000001c1 000001de 000001ed
0000640 000001fb 00000207 0000021b 0000022a
0000660 0000023d 0000024e 00000269 00000277
0000700 00000287 00000297 000002b6 000002ca
0000720 000002dc 000002f0 00000304 00000321
0000740 0000033e 0000035d 0000037a 00000395
0000760 000003ae 000003b6 000003be 000003c6
0001000 000003ce 000003dc 000003e9 000003f8
0001020
```

Il y a 132 valeurs, qui vaut 66*3. Peut-être qu'il y a deux valeurs 32-bit pour chaque symbole, mais peut-être y a-t-il deux tableaux? Voyons

Les valeurs débutant par `0x1000` peuvent être une adresse.

Ceci est un fichier .SYM pour une DLL après tout, et l'adresse de base par défaut des DLL win32 est `0x10000000`, et le code débute en général en `0x10001000`.

Lorsque nous ouvrons le fichier orawtc8.dll dans IDA, l'adresse de base est différente, mais néanmoins, la première fonction est:

```
.text:60351000 sub_60351000    proc near
.text:60351000
.text:60351000 arg_0    = dword ptr  8
.text:60351000 arg_4    = dword ptr  0Ch
.text:60351000 arg_8    = dword ptr  10h
.text:60351000
.text:60351000          push    ebp
.text:60351001          mov     ebp, esp
.text:60351003          mov     eax, dword_60353014
```

```
.text:60351008          cmp     eax, 0FFFFFFFFh
.text:6035100B          jnz     short loc_6035104F
.text:6035100D          mov     ecx, hModule
.text:60351013          xor     eax, eax
.text:60351015          cmp     ecx, 0FFFFFFFFh
.text:60351018          mov     dword_60353014, eax
.text:6035101D          jnz     short loc_60351031
.text:6035101F          call    sub_603510F0
.text:60351024          mov     ecx, eax
.text:60351026          mov     eax, dword_60353014
.text:6035102B          mov     hModule, ecx
.text:60351031
.text:60351031 loc_60351031:    ; CODE XREF: sub_60351000+1D
.text:60351031          test    ecx, ecx
.text:60351033          jbe     short loc_6035104F
.text:60351035          push    offset ProcName ; "ax_reg"
.text:6035103A          push    ecx             ; hModule
.text:6035103B          call    ds:GetProcAddress
...
```

Ouah, la chaîne «ax_reg» me dit quelque chose.

C'est en effet la première chaîne dans le bloc de chaîne! Donc le nom de cette fonction semble être «ax_reg».

La seconde fonction est:

```
.text:60351080 sub_60351080    proc near
.text:60351080
.text:60351080 arg_0    = dword ptr  8
.text:60351080 arg_4    = dword ptr  0Ch
.text:60351080
.text:60351080          push    ebp
.text:60351081          mov     ebp, esp
.text:60351083          mov     eax, dword_60353018
.text:60351088          cmp     eax, 0FFFFFFFFh
.text:6035108B          jnz     short loc_603510CF
.text:6035108D          mov     ecx, hModule
.text:60351093          xor     eax, eax
.text:60351095          cmp     ecx, 0FFFFFFFFh
.text:60351098          mov     dword_60353018, eax
.text:6035109D          jnz     short loc_603510B1
.text:6035109F          call    sub_603510F0
.text:603510A4          mov     ecx, eax
.text:603510A6          mov     eax, dword_60353018
.text:603510AB          mov     hModule, ecx
.text:603510B1
.text:603510B1 loc_603510B1:    ; CODE XREF: sub_60351080+1D
.text:603510B1          test    ecx, ecx
.text:603510B3          jbe     short loc_603510CF
.text:603510B5          push    offset aAx_unreg ; "ax_unreg"
.text:603510BA          push    ecx             ; hModule
.text:603510BB          call    ds:GetProcAddress
...
```

La chaîne «ax_unreg» est aussi la seconde chaîne dans le bloc de chaîne!

L'adresse de début de la seconde fonction est 0x60351080, et la seconde valeur dans le bloc binaire est 10001080. Donc ceci est l'adresse, mais pour une DLL avec l'adresse de base par défaut.

Nous pouvons rapidement vérifier et être sûr que les 66 premières valeurs dans le tableau (i.e., la première moitié du tableau) sont simplement les adresses des fonctions dans la DLL, incluant quelques labels, etc. Bien, qu'est-ce que l'autre partie du tableau? Les autres 66 valeurs qui commencent par 0x0000? Elles semblent être dans l'intervalle [0...0x3F8]. Et elles ne ressemblent pas à des champs de bits: la série de nombres augmente.

Le dernier chiffre hexadécimal semble être aléatoire, donc, il est peu probable que ça soit l'adresse de quelque chose (il serait divisible par 4 ou peut-être 8 ou 0x10 autrement).

Demandons-nous: qu'est-ce que les développeurs d'Oracle RDBMS pourraient avoir sauvegardé ici, dans ce fichier?

Supposition rapide: ça pourrait être l'adresse de la chaîne de texte (nom de la fonction).

Ça peut être vérifié rapidement, et oui, chaque nombre est simplement la position du premier caractère dans le bloc de chaînes.

Ça y est! C'est fini.

Nous allons écrire un utilitaire pour convertir ces fichiers .SYM en un script IDA, donc nous pourrons charger le script .idc et mettre les noms de fonction:

```
#include <stdio.h>
#include <stdint.h>
#include <io.h>
#include <assert.h>
#include <malloc.h>
#include <fcntl.h>
#include <string.h>

int main (int argc, char *argv[])
{
        uint32_t sig, cnt, offset;
        uint32_t *d1, *d2;
        int     h, i, remain, file_len;
        char    *d3;
        uint32_t array_size_in_bytes;

        assert (argv[1]); // file name
        assert (argv[2]); // additional offset (if needed)

        // additional offset
        assert (sscanf (argv[2], "%X", &offset)==1);

        // get file length
        assert ((h=open (argv[1], _O_RDONLY | _O_BINARY, 0))!=-1);
        assert ((file_len=lseek (h, 0, SEEK_END))!=-1);
```

```
        assert (lseek (h, 0, SEEK_SET)!=-1);

        // read signature
        assert (read (h, &sig, 4)==4);
        // read count
        assert (read (h, &cnt, 4)==4);

        assert (sig==0x4D59534F); // OSYM

        // skip timedatestamp (for 11g)
        //_lseek (h, 4, 1);

        array_size_in_bytes=cnt*sizeof(uint32_t);

        // load symbol addresses array
        d1=(uint32_t*)malloc (array_size_in_bytes);
        assert (d1);
        assert (read (h, d1, array_size_in_bytes)==array_size_in_bytes);

        // load string offsets array
        d2=(uint32_t*)malloc (array_size_in_bytes);
        assert (d2);
        assert (read (h, d2, array_size_in_bytes)==array_size_in_bytes);

        // calculate strings block size
        remain=file_len-(8+4)-(cnt*8);

        // load strings block
        assert (d3=(char*)malloc (remain));
        assert (read (h, d3, remain)==remain);

        printf ("#include <idc.idc>\n\n");
        printf ("static main() {\n");

        for (i=0; i<cnt; i++)
                printf ("\tMakeName(0x%08X, \"%s\");\n", offset + d1[i], &↙
    ↘ d3[d2[i]]);

        printf ("}\n");

        close (h);
        free (d1); free (d2); free (d3);
};
```

Voici un exemple de comment ça fonctionne:

```
#include <idc.idc>

static main() {
        MakeName(0x60351000, "_ax_reg");
        MakeName(0x60351080, "_ax_unreg");
        MakeName(0x603510F0, "_loaddll");
        MakeName(0x60351150, "_wtcsrin0");
        MakeName(0x60351160, "_wtcsrin");
```

```
        MakeName(0x603511C0, "_wtcsrfre");
        MakeName(0x603511D0, "_wtclkm");
        MakeName(0x60351370, "_wtcstu");
...
}
```

Les fichiers d'exemple qui ont été utilisés dans cet exemple sont ici: beginners.re.

Oh, essayons avec Oracle RDBMS pour win64. Les adresses devraient faire 64-bit, n'est-ce pas?

Le motif sur 8 octets est visible encore plus facilement ici:

```
Hiew: oracle.sym
   oracle.sym                               FRO --------                  00000000
00000000:  4F 53 59 4D-41 4D 36 34-BD 6D 05 00-00 00 00 00  OSYMAM64╜m
00000010:  CD 21 2A 47-00 00 00 00-00 00 00 00-00 00 00 00  =!*G
00000020:  00 00 00 00-00 00 00 00-00 40 00 00-00 00 00 00           @
00000030:  00 10 40 00-00 00 00 00-6C 10 40 00-00 00 00 00   @      l @
00000040:  04 11 40 00-00 00 00 00-80 13 40 00-00 00 00 00    @      A @
00000050:  E3 13 40 00-00 00 00 00-01 14 40 00-00 00 00 00  y @        @
00000060:  1F 14 40 00-00 00 00 00-3E 14 40 00-00 00 00 00    @      > @
00000070:  54 14 40 00-00 00 00 00-1E 18 40 00-00 00 00 00  T @        @
00000080:  97 1B 40 00-00 00 00 00-C1 1B 40 00-00 00 00 00  Ч @      ┴ @
00000090:  0A 1C 40 00-00 00 00 00-4C 1C 40 00-00 00 00 00    @      L @
000000A0:  7A 1C 40 00-00 00 00 00-98 1C 40 00-00 00 00 00  z @      Ш @
000000B0:  E7 25 40 00-00 00 00 00-11 26 40 00-00 00 00 00  ч%@       &@
000000C0:  80 26 40 00-00 00 00 00-C4 26 40 00-00 00 00 00  A&@      ─&@
000000D0:  F4 26 40 00-00 00 00 00-24 27 40 00-00 00 00 00  Ї&@      $'@
000000E0:  50 27 40 00-00 00 00 00-78 27 40 00-00 00 00 00  P'@      x'@
000000F0:  A0 27 40 00-00 00 00 00-4E 28 40 00-00 00 00 00  a'@      N(@
00000100:  26 29 40 00-00 00 00 00-B4 2C 40 00-00 00 00 00  &)@      ┤,@
00000110:  66 2D 40 00-00 00 00 00-A6 2D 40 00-00 00 00 00  f-@      ж-@
00000120:  30 2E 40 00-00 00 00 00-BA 2E 40 00-00 00 00 00  0.@      ║.@
00000130:  F2 30 40 00-00 00 00 00-84 31 40 00-00 00 00 00  Є0@      Д1@
00000140:  F0 31 40 00-00 00 00 00-5E 32 40 00-00 00 00 00  Ё1@      ^2@
00000150:  CC 32 40 00-00 00 00 00-3A 33 40 00-00 00 00 00  ╠2@      :3@
00000160:  A8 33 40 00-00 00 00 00-16 34 40 00-00 00 00 00  и3@       4@
00000170:  84 34 40 00-00 00 00 00-F2 34 40 00-00 00 00 00  Д4@      Є4@
00000180:  60 35 40 00-00 00 00 00-CC 35 40 00-00 00 00 00  `5@      ╠5@
00000190:  3A 36 40 00-00 00 00 00-A8 36 40 00-00 00 00 00  :6@      и6@
000001A0:  16 37 40 00-00 00 00 00-84 37 40 00-00 00 00 00   7@      Д7@
 1Global 2FilBlk 3CryBlk 4ReLoad 5       6String 7Direct 8Table 9       10Leave 11
```

Fig. 9.26: Exemple de fichier .SYM d'Oracle RDBMS pour win64

Donc oui, toutes les tables ont maintenant des éléments 64-bit, même les offsets de chaîne!

La signature est maintenant `OSYMAM64`, pour distinguer la plate-forme cible, apparemment.

C'est fini!

Voici aussi la bibliothèque qui a des fonctions pour accéder les fichiers .SYM d'Oracle RDBMS : GitHub.

9.6 Oracle RDBMS : fichiers .MSB-files

When working toward the solution of a problem, it always helps if you know the answer.

Murphy's Laws, Rule of Accuracy

Ceci est un fichier binaire qui contient les messages d'erreur avec leur numéro correspondant. Essayons de comprendre son format et de trouver un moyen de les extraire.

Il y a des fichiers de messages d'erreur d'Oracle RDBMS au format texte, donc nous pouvons comparer le texte et les fichiers binaires paqués[13].

Ceci est le début du fichier texte ORAUS.MSG avec des commentaires non pertinents supprimés:

Listing 9.11 : Beginning of ORAUS.MSG file without comments

```
00000, 00000, "normal, successful completion"
00001, 00000, "unique constraint (%s.%s) violated"
00017, 00000, "session requested to set trace event"
00018, 00000, "maximum number of sessions exceeded"
00019, 00000, "maximum number of session licenses exceeded"
00020, 00000, "maximum number of processes (%s) exceeded"
00021, 00000, "session attached to some other process; cannot switch ↙
    ↪ session"
00022, 00000, "invalid session ID; access denied"
00023, 00000, "session references process private memory; cannot detach ↙
    ↪ session"
00024, 00000, "logins from more than one process not allowed in single-↙
    ↪ process mode"
00025, 00000, "failed to allocate %s"
00026, 00000, "missing or invalid session ID"
00027, 00000, "cannot kill current session"
00028, 00000, "your session has been killed"
00029, 00000, "session is not a user session"
00030, 00000, "User session ID does not exist."
00031, 00000, "session marked for kill"
...
```

Le premier nombre est le code d'erreur. Le second contient peut-être des flags supplémentaires.

[13] Les fichiers texte open-source n'existent pas dans Oracle RDBMS pour chaque fichier .MSB, c'est donc pourquoi nous allons travailler sur leur format de fichier

Maintenant ouvrons le fichier binaire ORAUS.MSB et trouvons ces chaînes de teste. Et elles y sont:

Fig. 9.27: Hiew: first block

Nous voyons les chaînes de texte (celles du début du fichier ORAUS.MSG inclues) imbriquées avec d'autres sortes de valeurs binaire. Avec un examen rapide, nous pouvons voir que la partie principale du fichier binaire est divisée en bloc de taille 0x200 (512) octets.

Regardons le contenu du premier bloc:

```
Hiew: oraus.msb
   C:\tmp\oraus.msb                      FRO --------                    00001400
00001400:  0A 00 00 00-00 00 44 00-01 00 00 00-61 00 11 00        D     a
00001410:  00 00 83 00-12 00 00 00-A7 00 13 00-00 00 CA 00    Г     з     ╨
00001420:  14 00 00 00-F5 00 15 00-00 00 1E 01-16 00 00 00        ї
00001430:  5B 01 17 00-00 00 7C 01-18 00 00 00-BC 01 00 00   [     |     ╝
00001440:  00 00 00 02-6E 6F 72 6D-61 6C 2C 20-73 75 63 63      normal, succ
00001450:  65 73 73 66-75 6C 20 63-6F 6D 70 6C-65 74 69 6F   essful completio
00001460:  6E 75 6E 69-71 75 65 20-63 6F 6E 73-74 72 61 69   nunique constrai
00001470:  6E 74 20 28-25 73 2E 25-73 29 20 76-69 6F 6C 61   nt (%s.%s) viola
00001480:  74 65 64 73-65 73 73 69-6F 6E 20 72-65 71 75 65   tedsession reque
00001490:  73 74 65 64-20 74 6F 20-73 65 74 20-74 72 61 63   sted to set trac
000014A0:  65 20 65 76-65 6E 74 6D-61 78 69 6D-75 6D 20 6E   e eventmaximum n
000014B0:  75 6D 62 65-72 20 6F 66-20 73 65 73-73 69 6F 6E   umber of session
000014C0:  73 20 65 78-63 65 65 64-65 64 6D 61-78 69 6D 75   s exceededmaximu
000014D0:  6D 20 6E 75-6D 62 65 72-20 6F 66 20-73 65 73 73   m number of sess
000014E0:  69 6F 6E 20-6C 69 63 65-6E 73 65 73-20 65 78 63   ion licenses exc
000014F0:  65 65 64 65-64 6D 61 78-69 6D 75 6D-20 6E 75 6D   eededmaximum num
00001500:  62 65 72 20-6F 66 20 70-72 6F 63 65-73 73 65 73   ber of processes
00001510:  20 28 25 73-29 20 65 78-63 65 65 64-65 64 73 65    (%s) exceededse
00001520:  73 73 69 6F-6E 20 61 74-74 61 63 68-65 64 20 74   ssion attached t
00001530:  6F 20 73 6F-6D 65 20 6F-74 68 65 72-20 70 72 6F   o some other pro
00001540:  63 65 73 73-3B 20 63 61-6E 6E 6F 74-20 73 77 69   cess; cannot swi
00001550:  74 63 68 20-73 65 73 73-69 6F 6E 69-6E 76 61 6C   tch sessioninval
00001560:  69 64 20 73-65 73 73 69-6F 6E 20 49-44 3B 20 61   id session ID; a
00001570:  63 63 65 73-73 20 64 65-6E 69 65 64-73 65 73 73   ccess deniedsess
00001580:  69 6F 6E 20-72 65 66 65-72 65 6E 63-65 73 20 70   ion references p
00001590:  72 6F 63 65-73 73 20 70-72 69 76 61-74 65 20 6D   rocess private m
000015A0:  65 6D 6F 72-79 3B 20 63-61 6E 6E 6F-74 20 64 65   emory; cannot de
000015B0:  74 61 63 68-20 73 65 73-73 69 6F 6E-6C 6F 67 69   tach sessionlogi
 1Global 2FilBlk 3CryBlk 4ReLoad 5       6String 7Direct 8Table 9       10Leave 11
```

Fig. 9.28: Hiew: first block

Ici nous voyons les textes des premiers messages d'erreur. Ce que nous voyons aussi, c'est qu'il n'y a pas d'octet à zéro entre les messages d'erreur. Ceci implique que ce ne sont pas des chaînes C terminées par null. Par conséquent, la longueur de chaque message d'erreur doit être encodée d'une façon ou d'une autre. Essayons aussi de trouver le numéro d'erreur. Le fichier ORAUS.MSG débute par ceci: 0, 1, 17 (0x11), 18 (0x12), 19 (0x13), 20 (0x14), 21 (0x15), 22 (0x16), 23 (0x17), 24 (0x18)... Nous allons trouver ces nombres au début du bloc et les marquer avec des lignes rouge. La période entre les codes d'erreur est de 6 octets.

Ceci implique qu'il y a probablement 6 octets d'information alloués pour chaque message d'erreur.

La première valeur 16-bit (0xA ici ou 10) indique le nombre de messages dans

chaque bloc: ceci peut être vérifié en examinant d'autres blocs. En effet: les messages d'erreur ont une taille arbitraire. Certains sont plus long, d'autres plus court. Mais la taille du bloc est toujours fixée, ainsi, vous ne savez jamais combien de messages d'erreur sont stockés dans chaque bloc.

Comme nous l'avons déjà noté, puisqu'il ne s'agit pas de chaîne C terminée par null, leur taille doit être encodée quelque part. La taille de la première chaîne «normal, successful completion» est de 29 (0x1D) octets. La taille de la seconde chaîne «unique constraint (%s.%s) violated» est de 34 (0x22) octets. Nous ne trouvons pas ces valeurs (0x1D ou/et 0x22) dans le bloc.

Il y a aussi une autre chose. Oracle RDBMS doit déterminer la position de la chaîne qu'il y a besoin de charger dans le bloc, exact? La première chaîne «normal, successful completion» débute à la position 0x1444 (si nous comptons depuis le début du fichier) ou en 0x44 (depuis le début du bloc). La seconde chaîne «unique constraint (%s.%s) violated» débute à la position 0x1461 (depuis le début du fichier) ou en 0x61 (depuis le début du bloc). Ces nombres nous sont quelque peu familier! Nous pouvons clairement les voir au début du bloc.

Donc, chaque bloc de 6 octets est:

- 16-bit numéro d'erreur;
- 16-bit à zéro (peut-être des flags additionnels) ;
- position du début de la chaîne de texte dans le bloc courant.

Nous pouvons rapidement vérifier les autres valeurs et être sûr que notre supposition est correcte. Et il y a aussi le dernier bloc «factice» de 6 octets avec un numéro d'erreur à zéro et débutant après le dernier caractère du dernier message d'erreur. Peut-être est-ce ainsi que la longueur du texte du message est déterminée? Nous énumérons juste les blocs de 6 octets pour trouver le numéro d'erreur dont nous avons besoin, puis nous obtenons la position de la chaîne de texte, puis la longueur de la chaîne de texte en cherchant le bloc de 6 octets suivant! De cette façon nous déterminons les limites de la chaîne! Cette méthode nous permet d'économiser un peu d'espace en ne sauvegardant pas la taille de la chaîne de texte dans le fichier!

Il n'est pas possible de dire si ça sauve beaucoup d'espace, mais c'est un truc astucieux.

Revenons à l'entête de fichier .MSB:

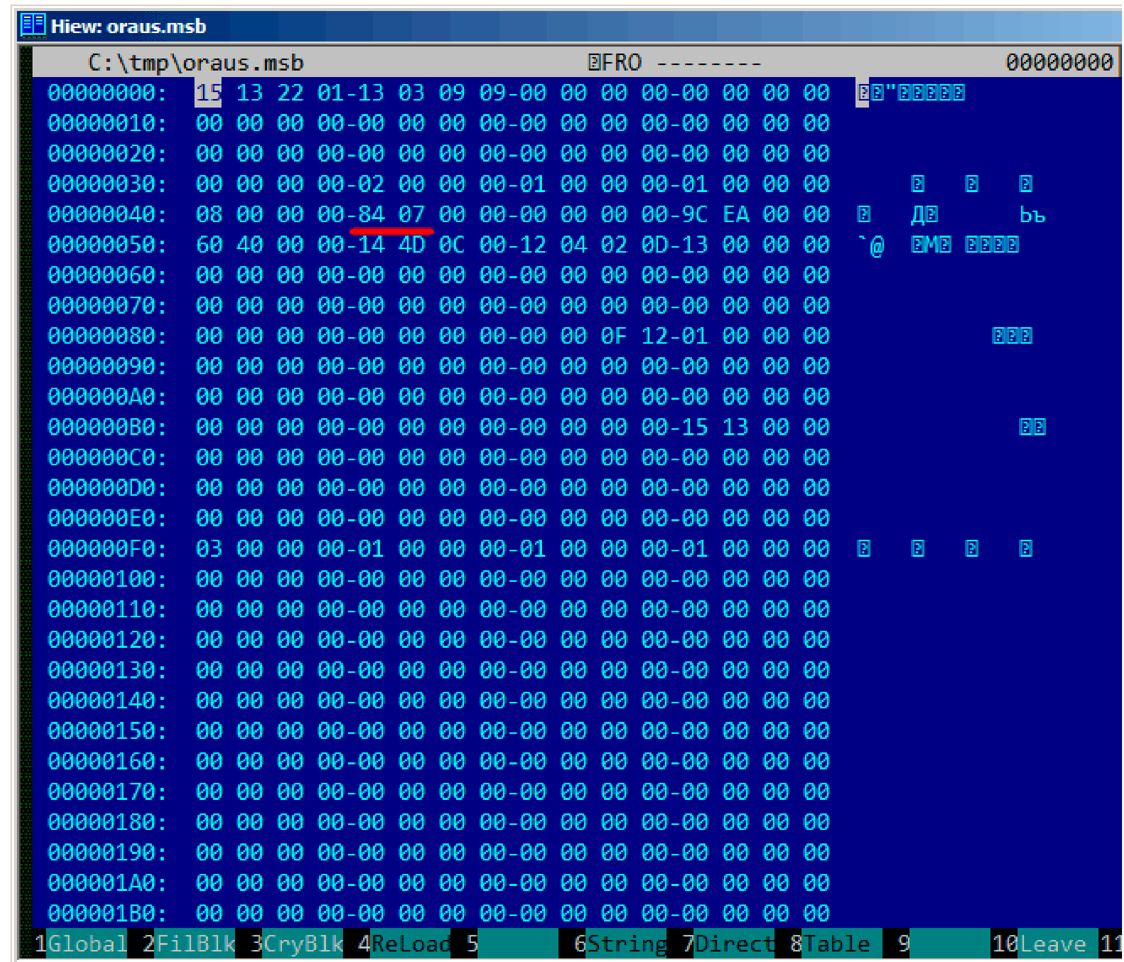


Fig. 9.29: Hiew: entête de fichier

Maintenant nous pouvons trouver rapidement le nombre de blocs dans le fichier (marqué en rouge). Nous pouvons vérifier d'autres fichiers .MSB et nous voyons que c'est vrai pour chacun d'entre eux.

Il y a de nombreuses autres valeurs, mais nous ne voulons pas les examiner, puisque notre job (un utilitaire d'extraction) est fait.

Si nous devions écrire un générateur de fichier .MSB, nous devrions probablement comprendre la signification des autres valeurs.

Il y a aussi une table qui vient après l'entête, qui contient probablement des valeurs 16-bit:

Fig. 9.30: Hiew: table last_errnos

Leurs tailles peuvent être déterminées visuellement (les lignes rouges sont dessinées ici).

En regardant ces valeurs, nous avons trouvé que chaque nombre 16-bit est le dernier code d'erreur pour chaque bloc.

C'est donc ainsi qu'Oracle RDBMS trouve rapidement le message d'erreur:

- charger la table que nous appellerons last_errnos (qui contient le dernier numéro d'erreur pour chaque bloc) ;
- trouver un bloc qui contient le numéro d'erreur que nous cherchons, en assumant que les codes d'erreur augmentent dans les blocs et aussi dans le fichier;

- charger le bloc spécifique;
- énumérer les structures de 6 octets jusqu'à trouver le numéro d'erreur;
- obtenir la position du premier du bloc de 6 octets courant;
- obtenir la position du dernier caractère du bloc de 6 octets suivant;
- charger tous les caractères du message dans cet intervalle.

Ceci est un programme C que nous avons écrit qui extrait les fichiers .MSB: beginners.re.

Voici aussi les deux fichiers que nous avons utilisé dans l'exemple (Oracle RDBMS 11.1.0.6) : beginners.re, beginners.re.

9.6.1 Résumé

Cette méthode est probablement désuète pour les ordinateurs moderne. Je suppose que ce format de fichier a été développé au milieu des années 80 par quelqu'un qui a aussi codé pour les *big iron*[14] en ayant à l'esprit l'économie d'espace et de mémoire. Néanmoins, ça a été une tâche intéressante mais facile de comprendre un format de fichier propriétaire sans regarder dans le code d'Oracle RDBMS.

9.7 Exercices

Essayez de rétro-ingénieurer tous les fichiers binaires de votre jeu favori, fichier des meilleurs scores inclus, ressources, etc.

Il y a aussi des fichiers binaires avec une structure connue: les fichiers utmp/wtmp, essayer de comprendre leur structure sans documentation.

L'entête EXIF dans les fichiers JPEG est documenté, mais vous pouvez essayer de comprendre sa structure sans aide, simplement en prenant des photos à différentes heures/dates, lieux, et essayer de trouver la date/heure et position GPS dans les données EXIF. Essayez de modifier la position GPS, uploadez le fichier JPEG dans Facebook et regardez, comment il va mettre votre photo sur la carte.

Essayez de patcher toutes les informations dans un fichier MP3 et voyez comment réagit votre lecteur de MP3 favori.

9.8 Pour aller plus loin

Pierre Capillon – Black-box cryptanalysis of home-made encryption algorithms: a practical case study.

How to Hack an Expensive Camera and Not Get Killed by Your Wife.

[14]NDT: Gros ordinateur de type mainframe.

Chapitre 10

Dynamic binary instrumentation

Les outils DBI peuvent être vus comme des débogueurs très avancés et rapide.

10.1 Utiliser PIN DBI pour intercepter les XOR

PIN d'Intel est un outil DBI. Cela signifie qu'il prend un binaire compilé et y insère vos instructions, où vous voulez.

Essayons d'intercepter toutes les instructions XOR. Elles sont utilisées intensément en cryptographie, et nous pouvons essayer de lancer l'archiveur WinRAR en mode chiffrement avec l'espoir que des instructions sont effectivement utilisées durant le chiffrement.

Voici le code source de mon outil PIN: `https://beginners.re/paywall/RE4B-source/current-tree//DBI/XOR/files//XOR_ins.cpp`.

Le code est presque auto-documenté: il scanne le fichier exécutable en entrée à la recherche des instructions XOR/PXOR et insère un appel à notre fonction avant chaque. La fonction log_info() vérifie d'abord si les opérandes sont différents (puisque l'instruction XOR est souvent utilisée pour effacer simplement un registre, comme XOR EAX, EAX), et si ils sont différents, il incrémente un compteur à cette EIP/RIP, afin que les statistiques soient collectées.

J'ai préparé deux fichiers pour tester: test1.bin (30720 octets) et test2.bin (5547752 octets), je vais les compresser avec RAR avec un mot de passe et voir les différences dans les statistiques.

Vous devez aussi désactiver ASLR [1], afin que l'outil PIN rapporte les mêmes RIPs que dans l'exécutable RAR.

Maintenant, lançons-le:

[1] `https://stackoverflow.com/q/9560993`

```
c:\pin-3.2-81205-msvc-windows\pin.exe -t XOR_ins.dll -- rar a -↙
    ↳ pLongPassword tmp.rar test1.bin
c:\pin-3.2-81205-msvc-windows\pin.exe -t XOR_ins.dll -- rar a -↙
    ↳ pLongPassword tmp.rar test2.bin
```

Maintenant voici les statistiques pour test1.bin:
https://beginners.re/paywall/RE4B-source/current-tree//DBI/XOR/files//XOR_ins.out.test1. ... et pour test2.bin:
https://beginners.re/paywall/RE4B-source/current-tree//DBI/XOR/files//XOR_ins.out.test2. Jusqu'ici, vous pouvez ignorer toutes les adresses autres que ip=0x1400xxxxx, qui sont dans d'autres DLLs.

Maintenant, regardons la différence: https://beginners.re/paywall/RE4B-source/current-tree//DBI/XOR/files//XOR_ins.diff.

Certaines instructions XOR sont exécutées plus souvent pour test2.bin (qui est plus gros) que pour test1.bin (qui est plus petit). Donc elles sont clairement liées à la taille du fichier!

Le premier bloc de différence est:

```
< ip=0x140017b21 count=0xd84
< ip=0x140017b48 count=0x81f
< ip=0x140017b59 count=0x858
< ip=0x140017b6a count=0xc13
< ip=0x140017b7b count=0xefc
< ip=0x140017b8a count=0xefd
< ip=0x140017b92 count=0xb86
< ip=0x140017ba1 count=0xf01
---
> ip=0x140017b21 count=0x9eab5
> ip=0x140017b48 count=0x79863
> ip=0x140017b59 count=0x862e8
> ip=0x140017b6a count=0x99495
> ip=0x140017b7b count=0xa891c
> ip=0x140017b8a count=0xa89f4
> ip=0x140017b92 count=0x8ed72
> ip=0x140017ba1 count=0xa8a8a
```

C'est en effet une sorte de boucle à l'intérieur de RAR.EXE:

```
.text:0000000140017B21 loc_140017B21:
.text:0000000140017B21                 xor     r11d, [rbx]
.text:0000000140017B24                 mov     r9d, [rbx+4]
.text:0000000140017B28                 add     rbx, 8
.text:0000000140017B2C                 mov     eax, r9d
.text:0000000140017B2F                 shr     eax, 18h
.text:0000000140017B32                 movzx   edx, al
.text:0000000140017B35                 mov     eax, r9d
.text:0000000140017B38                 shr     eax, 10h
.text:0000000140017B3B                 movzx   ecx, al
.text:0000000140017B3E                 mov     eax, r9d
.text:0000000140017B41                 shr     eax, 8
```

```
.text:0000000140017B44                 mov     r8d, [rsi+rdx*4]
.text:0000000140017B48                 xor     r8d, [rsi+rcx*4+400h]
.text:0000000140017B50                 movzx   ecx, al
.text:0000000140017B53                 mov     eax, r11d
.text:0000000140017B56                 shr     eax, 18h
.text:0000000140017B59                 xor     r8d, [rsi+rcx*4+800h]
.text:0000000140017B61                 movzx   ecx, al
.text:0000000140017B64                 mov     eax, r11d
.text:0000000140017B67                 shr     eax, 10h
.text:0000000140017B6A                 xor     r8d, [rsi+rcx*4+1000h]
.text:0000000140017B72                 movzx   ecx, al
.text:0000000140017B75                 mov     eax, r11d
.text:0000000140017B78                 shr     eax, 8
.text:0000000140017B7B                 xor     r8d, [rsi+rcx*4+1400h]
.text:0000000140017B83                 movzx   ecx, al
.text:0000000140017B86                 movzx   eax, r9b
.text:0000000140017B8A                 xor     r8d, [rsi+rcx*4+1800h]
.text:0000000140017B92                 xor     r8d, [rsi+rax*4+0C00h]
.text:0000000140017B9A                 movzx   eax, r11b
.text:0000000140017B9E                 mov     r11d, r8d
.text:0000000140017BA1                 xor     r11d, [rsi+rax*4+1C00h]
.text:0000000140017BA9                 sub     rdi, 1
.text:0000000140017BAD                 jnz     loc_140017B21
```

Que fait-elle? Aucune idée à ce stade.

La suivante:

```
< ip=0x14002c4f1 count=0x4fce
---
> ip=0x14002c4f1 count=0x4463be
```

0x4fce est 20430, qui est proche de la taille de test1.bin (30720 octets). 0x4463be est 4481982, qui est proche de la taille de test2.bin (5547752 octets). Par égal, mais proche.

Ceci est un morceau de code avec cette instruction XOR:

```
.text:000000014002C4EA loc_14002C4EA:
.text:000000014002C4EA                 movzx   eax, byte ptr [r8]
.text:000000014002C4EE                 shl     ecx, 5
.text:000000014002C4F1                 xor     ecx, eax
.text:000000014002C4F3                 and     ecx, 7FFFh
.text:000000014002C4F9                 cmp     [r11+rcx*4], esi
.text:000000014002C4FD                 jb      short loc_14002C507
.text:000000014002C4FF                 cmp     [r11+rcx*4], r10d
.text:000000014002C503                 ja      short loc_14002C507
.text:000000014002C505                 inc     ebx
```

Le corps de la boucle peut être écrit comme:

```
state = input_byte ^ (state<<5) & 0x7FFF}.
```

state est ensuite utilisé comme un index dans une table. Est-ce une sorte de CRC[2] ? Je ne sais pas, mais ça pourrait être une routine effectuant une somme de contrôle. Ou peut-être une routine CRC optimisée? Une idée?

Le bloc suivant:

```
< ip=0x14004104a count=0x367
< ip=0x140041057 count=0x367
---
> ip=0x14004104a count=0x24193
> ip=0x140041057 count=0x24193
```

```
.text:0000000140041039 loc_140041039:
.text:0000000140041039                 mov     rax, r10
.text:000000014004103C                 add     r10, 10h
.text:0000000140041040                 cmp     byte ptr [rcx+1], 0
.text:0000000140041044                 movdqu  xmm0, xmmword ptr [rax]
.text:0000000140041048                 jz      short loc_14004104E
.text:000000014004104A                 pxor    xmm0, xmm1
.text:000000014004104E
.text:000000014004104E loc_14004104E:
.text:000000014004104E                 movdqu  xmm1, xmmword ptr [rcx+18h]
.text:0000000140041053                 movsxd  r8, dword ptr [rcx+4]
.text:0000000140041057                 pxor    xmm1, xmm0
.text:000000014004105B                 cmp     r8d, 1
.text:000000014004105F                 jle     short loc_14004107C
.text:0000000140041061                 lea     rdx, [rcx+28h]
.text:0000000140041065                 lea     r9d, [r8-1]
.text:0000000140041069
.text:0000000140041069 loc_140041069:
.text:0000000140041069                 movdqu  xmm0, xmmword ptr [rdx]
.text:000000014004106D                 lea     rdx, [rdx+10h]
.text:0000000140041071                 aesenc  xmm1, xmm0
.text:0000000140041076                 sub     r9, 1
.text:000000014004107A                 jnz     short loc_140041069
.text:000000014004107C
```

Ce morceau possède les instructions PXOR et AESENC (la dernière est une instruction de chiffrement AES[3]). Donc oui, nous avons trouvé une fonction de chiffrement, RAR utilise AES.

Il y a ensuite un autre gros bloc d'instructions XOR presque contigus:

```
< ip=0x140043e10 count=0x23006
---
> ip=0x140043e10 count=0x23004
499c510
< ip=0x140043e56 count=0x22ffd
---
> ip=0x140043e56 count=0x23002
```

[2]Cyclic redundancy check
[3]Advanced Encryption Standard

Mais le compteur n'est pas très différent pendant la compression/chiffrement de test1.bin/test2.bin. Qu'y a-t-il à ces adresses?

```
.text:0000000140043E07                  xor     ecx, r9d
.text:0000000140043E0A                  mov     r11d, eax
.text:0000000140043E0D                  and     ecx, r10d
.text:0000000140043E10                  xor     ecx, r8d
.text:0000000140043E13                  rol     eax, 8
.text:0000000140043E16                  and     eax, esi
.text:0000000140043E18                  ror     r11d, 8
.text:0000000140043E1C                  add     edx, 5A827999h
.text:0000000140043E22                  ror     r10d, 2
.text:0000000140043E26                  add     r8d, 5A827999h
.text:0000000140043E2D                  and     r11d, r12d
.text:0000000140043E30                  or      r11d, eax
.text:0000000140043E33                  mov     eax, ebx
```

Googlons la constante 5A827999h... ceci ressemble à du SHA-1! mais pourquoi RAR utiliserait-il SHA-1 pendant le chiffrement?

Voici la réponse:

```
In comparison, WinRAR uses its own key derivation scheme that requires (↙
    ↳ password length * 2 + 11)*4096 SHA-1 transformations. 'Thats why it ↙
    ↳ takes longer to brute-force attack encrypted WinRAR archives.
```

(http://www.tomshardware.com/reviews/password-recovery-gpu,2945-8.html)

C'est la génération de la clef: le mot de passe entré est hashé plusieurs fois et le hash est utilisé comme clef AES. C'est pourquoi nous voyons que le comptage de l'instruction XOR est presque inchangé lorsque nous passons au fichier de test plus gros.

C'est tout ce qu'il faut faire, ça m'a pris quelques heures d'écrire cet outil et d'obtenir au moins 3 éléments: 1) c'est probablement une somme de contrôle; 2) chiffrement AES; 3) calcul de somme SHA-1. La première fonction est encore un mystère pour moi.

Cependant, ceci est impressionnant, car je ne me suis pas plongé dans le code de RAR (qui est propriétaire, bien sûr). Je n'ai même pas jeté un coup d'œil dans le code source de UnRAR (qui est disponible).

Les fichiers, incluant les fichiers de test et l'exécutable RAR que j'ai utilisé (win64, 5.40) : https://beginners.re/paywall/RE4B-source/current-tree//DBI/XOR/files/.

10.2 Cracker Minesweeper avec PIN

Dans ce livre, j'ai expliqué comment cracker Minesweeper pour Windows XP: 8.4 on page 1047.

Le Minesweeper de Windows Vista et 7 est différent: il a probablement été (r)écrit en C++, et l'information de la case n'est maintenant plus stockée dans un tableau global, mais plutôt dans des blocs du heap alloués par malloc.

Ceci est un cas où nous pouvons essayer l'outil PIN DBI.

10.2.1 Intercepter tous les appels à rand()

Tout d'abord, puisque Minesweeper dispose les mines aléatoirement, il doit appeler rand() ou une fonction similaire. Essayons d'intercepter tous les appels à rand() : https://beginners.re/paywall/RE4B-source/current-tree//DBI/minesweeper/minesweeper1.cpp.

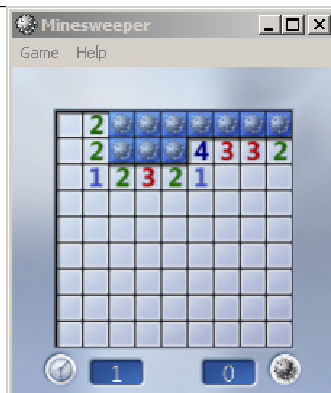
Nous pouvons maintenant le lancer:

```
c:\pin-3.2-81205-msvc-windows\pin.exe -t minesweeper1.dll -- C:\PATH\TO\↙
    ↘ MineSweeper.exe
```

Durant le démarrage, PIN cherche tous les appels à la fonction rand() et ajoute un hook juste après chaque appel. Le hook est la fonction RandAfter() que nous avons défini: elle logue la valeur et l'adresse de retour. Voici un log que j'ai obtenu en lançant la configuration 9*9 standard (10 mines) : https://beginners.re/paywall/RE4B-source/current-tree//DBI/minesweeper/minesweeper1.out.10mines. La fonction rand() a été appelée de nombreuses fois depuis différents endroits, mais a été appelée depuis 0x10002770d exactement 10 fois. J'ai changé la configuration de Minesweeper à 16*16 (40 mines) et rand() a été appelée 40 fois depuis 0x10002770d. Donc oui, c'est ce que l'on cherche. Lorsque je charge minesweeper.exe (depuis Windows 7) dans IDA et une fois que le PDB est récupéré depuis le site web de Microsoft, la fonction qui appelle rand() en 0x10002770d est appelée Board::placeMines().

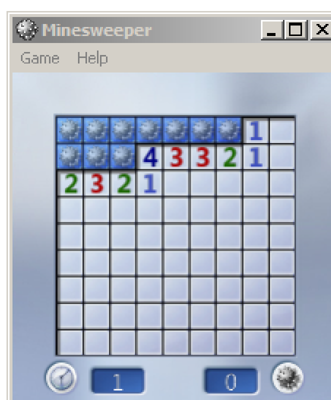
10.2.2 Remplacer les appels à rand() par notre function

Essayons maintenant de remplacer la fonction rand() avec notre version, qui renvoie toujours zéro: https://beginners.re/paywall/RE4B-source/current-tree//DBI/minesweeper/minesweeper2.cpp. Durant le démarrage, PIN remplace tous les appels à la fonction rand() par des appels à notre fonction, qui écrit dans le log et renvoie zéro. Ok, je l'ai lancé et ai cliqué sur la case la plus en haut à gauche.



Oui, contrairement à Minesweeper de Windows XP, les mines sont placées aléatoirement *après* que l'utilisateur ai cliqué sur une case, afin de garantir qu'il n'y a pas de mine sur la première case cliquée par l'utilisateur. Donc Minesweeper a placé les mines dans des cases autres que celle la plus en haut à gauche (où j'ai cliqué).

Maintenant j'ai cliqué sur la case la plus en haut à droite:



Ceci peut-être une sorte de blague? Je ne sais pas.

J'ai cliqué sur la 5ème case (droite du milieu) de la 1ère ligne:

C'est bien, car Minesweeper peut effectuer un placement correct même avec un PRNG aussi mauvais!

10.2.3 Regarder comment les mines sont placées

Comment pouvons-nous obtenir des informations sur où les mines sont placées? Le résultat de rand() semble être inutile: elle renvoie zéro à chaque fois, mais Minesweeper a réussi à placer les mines dans des cases différentes, quoique, alignées.

Ce Minesweeper est aussi écrit dans la tradition C++, donc il n'a pas de tableau global.

Mettons-nous dans la peau du programmeur. Il doit y avoir une boucle comme:

```
for (int i; i<mines_total; i++)
{
        // get coordinates using rand()
        // put a cell: in other words, modify a block allocated in heap
};
```

Comment pouvons-nous obtenir des information sur le bloc de qui est modifié à la 2nde étape? Ce que nous devons faire: 1) suivre toutes les allocations dans la heap en interceptant malloc()/realloc()/free(). 2) suivre toutes les écritures en mémoire (lent). 3) suivre les appels à rand().

Maintenant l'algorithme: 1) suivre tous les blocs du heap qui sont modifiés entre le 1er et le 2nd appel à rand() depuis 0x10002770d; 2) à chaque fois qu'un bloc du heap est libéré, afficher son contenu.

Suivre toutes les écritures en mémoire est lent, mais après le 2nd appel à rand(), nous n'avons plus besoin de les suivre (puisque nous avons déjà obtenu une liste de blocs intéressants à ce point), donc nous arrêtons.

Maintenant le code: `https://beginners.re/paywall/RE4B-source/current-tree//DBI/minesweeper/minesweeper3.cpp`.

Il s'avère que seulement 4 blocs de heap sont modifiés entre les deux premiers appels à rand(), voici à quoi ils ressemblent:

```
free(0x20aa6360)
free(): we have this block in our records, size=0x28
0x20AA6360: 36 00 00 00 4E 00 00 00-2D 00 00 00 29 00 00 00 "6...N...-...)↙
    ↳ ..."
0x20AA6370: 06 00 00 00 37 00 00 00-35 00 00 00 19 00 00 00 ↙
    ↳ "....7...5......."
0x20AA6380: 46 00 00 00 0B 00 00 00-                        "F.......       ↙
    ↳   "

...

free(0x20af9d10)
free(): we have this block in our records, size=0x18
0x20AF9D10: 0A 00 00 00 0A 00 00 00-0A 00 00 00 00 00 00 00 ↙
    ↳ "................"
0x20AF9D20: 60 63 AA 20 00 00 00 00-                        "`c. ....       ↙
    ↳   "

...

free(0x20b28b20)
free(): we have this block in our records, size=0x140
0x20B28B20: 02 00 00 00 03 00 00 00-04 00 00 00 05 00 00 00 ↙
    ↳ "................"
0x20B28B30: 07 00 00 00 08 00 00 00-0C 00 00 00 0D 00 00 00 ↙
    ↳ "................"
0x20B28B40: 0E 00 00 00 0F 00 00 00-10 00 00 00 11 00 00 00 ↙
    ↳ "................"
0x20B28B50: 12 00 00 00 13 00 00 00-14 00 00 00 15 00 00 00 ↙
    ↳ "................"
0x20B28B60: 16 00 00 00 17 00 00 00-18 00 00 00 1A 00 00 00 ↙
    ↳ "................"
0x20B28B70: 1B 00 00 00 1C 00 00 00-1D 00 00 00 1E 00 00 00 ↙
    ↳ "................"
0x20B28B80: 1F 00 00 00 20 00 00 00-21 00 00 00 22 00 00 00 ".... ↙
    ↳ ...!..."..."
0x20B28B90: 23 00 00 00 24 00 00 00-25 00 00 00 26 00 00 00 "#...\$↙
    ↳ ...%...&..."
0x20B28BA0: 27 00 00 00 28 00 00 00-2A 00 00 00 2B 00 00 00 ↙
    ↳ "'...(...*...+..."
0x20B28BB0: 2C 00 00 00 2E 00 00 00-2F 00 00 00 30 00 00 00 ↙
    ↳ ",......./...0..."
0x20B28BC0: 31 00 00 00 32 00 00 00-33 00 00 00 34 00 00 00 ↙
    ↳ "1...2...3...4..."
0x20B28BD0: 38 00 00 00 39 00 00 00-3A 00 00 00 3B 00 00 00 ↙
    ↳ "8...9...:...;..."
0x20B28BE0: 3C 00 00 00 3D 00 00 00-3E 00 00 00 3F 00 00 00 ↙
    ↳ "<...=...>...?..."
0x20B28BF0: 40 00 00 00 41 00 00 00-42 00 00 00 43 00 00 00 "@...A...B...C↙
    ↳ ..."
0x20B28C00: 44 00 00 00 45 00 00 00-47 00 00 00 48 00 00 00 "D...E...G...H↙
    ↳ ..."
0x20B28C10: 49 00 00 00 4A 00 00 00-4B 00 00 00 4C 00 00 00 "I...J...K...L↙
```

```
    ↪ ..."
0x20B28C20: 4D 00 00 00 4F 00 00 00-50 00 00 00 50 00 00 00 "M...O...P...P↙
    ↪ ..."
0x20B28C30: 50 00 00 00 50 00 00 00-50 00 00 00 50 00 00 00 "P...P...P...P↙
    ↪ ..."
0x20B28C40: 50 00 00 00 50 00 00 00-50 00 00 00 50 00 00 00 "P...P...P...P↙
    ↪ ..."
0x20B28C50: 50 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 "P↙
    ↪ ..............."

...

free(0x20af9cf0)
free(): we have this block in our records, size=0x18
0x20AF9CF0: 43 00 00 00 50 00 00 00-10 00 00 00 20 00 74 00 "C...P....... .↙
    ↪ t."
0x20AF9D00: 20 8B B2 20 00 00 00 00-                         " .. ....      ↙
    ↪   "
```

Nous voyons facilement que les plus gros blocs (avec une taille de 0x28 et 0x140) sont juste des tableaux de valeurs jusqu'à ≈ 0x50. Attendez... 0x50 est 80 en représentation décimale. et 9*9=81 (configuration standard de Minesweeper).

Après une rapide investigation, j'ai trouvé que chaque élément 32-bit est en fait les coordonnées d'une case. Une case est représentée en utilisant un seul nombre, c'est un nombre dans un tableau-2D. Ligne et colonne de chaque mine sont décodées comme ceci: *row=n / WIDTH; col=n % HEIGHT;*

Lorsque j'ai essayé de décoder ces deux blocs les plus gros, j'ai obtenu ces cartes de case:

```
try_to_dump_cells(). unique elements=0xa
......*..
..*......
.......*.
.........
.....*...
*.......*
**.......
.......*.
......*..

...

try_to_dump_cells(). unique elements=0x44
*.****.**
...******
*******.*
*********
*****.***
.*******.
..*******
*******.*
******.**
```

Il semble que le premier bloc soit juste une liste des mines placées, tandis que le second bloc est une liste des cases libres, mais le second semble quelque peu désynchroniser du premier, et une version inversée du premier ne coïncide que partiellement. Néanmoins, la première carte est correcte - nous pouvons jeter un coup d'œil dans le fichier de log alors que Minesweeper est encore chargé et presque toutes les cases sont cachées, et cliquer tranquillement sur les cases marquées d'un point ici.

Il semble donc que lorsque l'utilisateur clique pour l première fois quelque part, Minesweeper place les 10 mines, puis détruit le bloc avec leurs liste (peut-être copie-t-il toutes les données dans un autre bloc avant?), donc nous pouvons les voir lors de l'appel à free().

Un autre fait: la méthode Array<NodeType>::Add(NodeType) modifie les blocs que nous avons observé, et est appelée depuis de nombreux endroits, Board::placeMines() incluse. Mais c'est cool: je ne suis jamais allé dans les détails, tout a été résolu simplement en utilisant PIN.

Les fichiers: `https://beginners.re/paywall/RE4B-source/current-tree//DBI/minesweeper`.

10.2.4 Exercice

Essayez de comprendre comment le résultat de rand() est converti en coordonnée(s). Pour blaguer, faite que rand() renvoie des résultats tels que les mines soient placées en formant un symbole ou une figure.

10.3 Compiler Pin

Compiler Pin pour Windows peut s'avérer délicat. Ceci est ma recette qui fonctionne.

- Décompacter le dernier Pin, disons, `C:\pin-3.7\`
- Installer le dernier Cygwin, dans, disons, `c:\cygwin64`
- Installer MSVC 2015 ou plus récent.
- Ouvrir le fichier `C:\pin-3.7\source\tools\Config\makefile.default.rules`, remplacer `mkdir -p $@` par `/bin/mkdir -p $@`
- (Si nécessaire) dans `C:\pin-3.7\source\tools\SimpleExamples\makefile.rules`, ajouter votre pintool à la liste TEST_TOOL_ROOTS.
- Ouvrir "VS2015 x86 Native Tools Command Prompt". Taper:

```
cd c:\pin-3.7\source\tools\SimpleExamples
c:\cygwin64\bin\make all TARGET=ia32
```

  Maintenant les outils pintools sont dans `c:\pin-3.7\source\tools\SimpleExamples\obj-ia32`
- Pour winx64, utiliser "x64 Native Tools Command Prompt" et lancer:

```
c:\cygwin64\bin\make all TARGET=intel64
```

- Lancer pintool:

```
c:\pin-3.7\pin.exe -t C:\pin-3.7\source\tools\SimpleExamples\obj-ia32\⤦
    ↳ XOR_ins.dll -- program.exe arguments
```

10.4 Pourquoi “instrumentation”?

Peut-être que c’est un terme de profilage de code. Il y a au moins deux méthodes: 1) ”échantillonnage”: vous rentrez dans le code se déroulant autant de fois que possible (des centaines par seconde), et regardez, où en est l’exécution à ce moment; 2) ”instrumentation”: le code compilé est intercalé avec de l’autre code, qui peut incrémenter des compteurs, etc.

Peut-être que les outils DBI ont hérités du terme?

Chapitre 11

Autres sujets

11.1 Utiliser IMUL au lieu de MUL

Un exemple comme listado.3.23.2 où deux valeurs non signées sont multipliées compile en listado.3.23.2 où IMUL est utilisé à la place de MUL.

Ceci est une propriété importante des instructions MUL et IMUL. Tout d'abord, les deux produisent une valeur 64-bit si deux valeurs 32-bit sont multipliées, ou une valeur 128-bit si deux valeurs 64-bit sont multipliées (le plus grand produit dans un environnement 32-bit est
0xffffffff*0xffffffff=0xfffffffe00000001). Mais les standards C/C++ n'ont pas de moyen d'accèder à la moitié supérieure du résultat, et un produit a toujours la même taille que ses multiplicandes. Et les deux instructions MUL et IMUL fonctionnent de la même manière si la moitié supérieure est ignorée, i.e, elles produisent le même résultat dans la partie inférieure. Ceci est une propriété importante de la façon de représenter les nombre en «complément à deux».

Donc, le compilateur C/C++ peut utiliser indifféremment ces deux instructions.

Mais IMUL est plus flexible que MUL, car elle prend n'importe quel(s) registre(s) comme source, alors que MUL nécessite que l'un des multiplicandes soit stocké dans le registre AX/EAX/RAX Et même plus que ça: MUL stocke son résultat dans la paire EDX:EAX en environnement 32-bit, ou RDX:RAX dans un 64-bit, donc elle calcule toujours le résultat complet. Au contraire, il est possible de ne mettre qu'un seul registre de destination lorsque l'on utilise IMUL, au lieu d'une paire, et alors le CPU calculera seulement la partie basse, ce qui fonctionne plus rapidement [voir Torborn Granlund, *Instruction latencies and throughput for AMD and Intel x86 processors*[1]].

Cela étant considéré, les compilateurs C/C++ peuvent générer l'instruction IMUL plus souvent que MUL.

Néanmoins, en utilisant les fonctions intrinsèques du compilateur, il est toujours possible d'effectuer une multiplication non signée et d'obtenir le résultat *complet*. Ceci est parfois appelé *multiplication étendue*. MSVC a une fonction intrinsèque pour ceci,

[1] http://yurichev.com/mirrors/x86-timing.pdf

appelée *__emul*[2] et une autre: *_umul128*[3]. GCC offre le type de données *__int128*, et dans le cas de multiplicandes 64-bit, ils sont déjà promus en 128-bit, puis le produit est stocké dans une autre valeur *__int128*, puis le résultat est décalé de 64 bits à droite, et vous obtenez la moitié haute du résultat[4].

11.1.1 Fonction MulDiv() dans Windows

Windows possède la fonction MulDiv()[5], fonction qui fusionne une multiplication et une division, elle multiplie deux entiers 32-bit dans une valeur 64-bit intermédiaire et la divise par un troisième entier 32-bit. C'est plus facile que d'utiliser deux fonctions intrinsèques, donc les développeurs de Microsoft ont écrit une fonction spéciale pour cela. Et il semble que ça soit une fonction très utilisée, à en juger par son utilisation.

11.2 Modification de fichier exécutable

11.2.1 code x86

Les tâches de modification courantes sont:

- Une des tâches la plus fréquente est de désactiver une instruction en l'écrasant avec des octets `0x90` (NOP).
- Les branchements conditionnels qui utilisent un code instruction tel que `74 xx` (JZ), peuvent être réécrits avec deux instructions NOP.

  Une autre technique consiste à désactiver un branchement conditionnel en écrasant le second octet avec la valeur 0 (*jump offset*).
- Une autre tâche courante consiste à faire en sorte qu'un branchement conditionnel soit effectué systématiquement. On y parvient en remplaçant le code instruction par `0xEB` qui correspond à l'instruction JMP.
- L'exécution d'une fonction peut être désactivée en remplaçant le premier octet par RETN (0xC3). Les fonctions dont la convention d'appel est `stdcall` (6.1.2 on page 953) font exception. Pour les modifier, il faut déterminer le nombre d'arguments (par exemple en trouvant une instruction RETN au sein de la fonction), puis en utilisant l'instruction RETN accompagnée d'un argument sur deux octets (0xC2).
- Il arrive qu'une fonction que l'on a désactivée doive retourner une valeur 0 ou 1. Certes on peut utiliser `MOV EAX, 0` ou `MOV EAX, 1`, mais cela occupe un peu trop d'espace.
  Une meilleure approche consiste à utiliser `XOR EAX, EAX` (2 octets `0x31 0xC0`) ou `XOR EAX, EAX / INC EAX` (3 octets `0x31 0xC0 0x40`).

Un logiciel peut être protégé contre les modifications. Le plus souvent la protection consiste à lire le code du programme (en mémoire) et à en calculer une valeur de

[2] https://msdn.microsoft.com/en-us/library/d2s81xt0(v=vs.80).aspx
[3] https://msdn.microsoft.com/library/3dayytw9%28v=vs.100%29.aspx
[4] Exemple: http://stackoverflow.com/a/13187798
[5] https://msdn.microsoft.com/en-us/library/windows/desktop/aa383718(v=vs.85).aspx

contrôle. Cette technique nécessite que la protection lise le code avant de pouvoir agir. Elle peut donc être détectée en positionnant un point d'arrêt déclenché par la lecture de la mémoire contenant le code.

tracer possède l'option BPM pour ce faire.

La partie du fichier au format PE qui contient les informations de relogement (6.5.2 on page 989) ne doivent pas être modifiées par les patchs car le chargeur Windows risquerait d'écraser les modifications apportées. (Ces parties sont présentées sous forme grisées dans Hiew, par exemple: fig.1.21).

En dernier ressort, il est possible d'effectuer des modifications qui contournent les relogements, ou de modifier directement la table des relogements.

11.3 Statistiques sur le nombre d'arguments d'une fonction

J'ai toujours été intéressé par le nombre moyen d'arguments d'une fonction.

J'ai donc analysé un bon nombre de DLLs 32 bits de Windows7 (crypt32.dll, mfc71.dll, msvcr100.dll, shell32.dll, user32.dll, d3d11.dll, mshtml.dll, msxml6.dll, sqlncli11.dll, wininet.dll, mfc120.dll, msvbvm60.dll, ole32.dll, themeui.dll, wmp.dll) (parce qu'elles utilisent la convention d'appel *stdcall* et qu'il est donc facile de retrouver les instructions `RETN X` en utilisant la commande *grep* sur leur code une fois celui-ci désassemblé).

- no arguments: ≈ 29%
- 1 argument: ≈ 23%
- 2 arguments: ≈ 20%
- 3 arguments: ≈ 11%
- 4 arguments: ≈ 7%
- 5 arguments: ≈ 3%
- 6 arguments: ≈ 2%
- 7 arguments: ≈ 1%

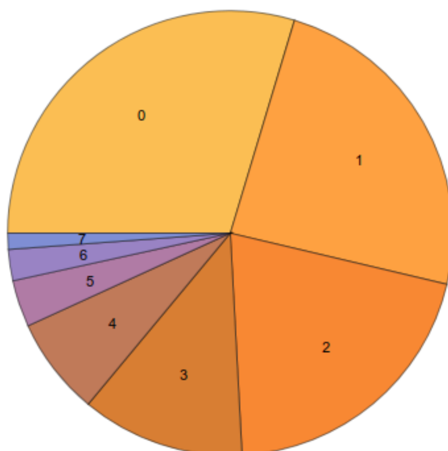


Fig. 11.1: Statistiques du nombre d'arguments moyen d'une fonction

Ces nombres dépendent beaucoup du style de programmation et peuvent s'avérer très différents pour d'autres logiciels.

11.4 Fonctions intrinsèques du compilateur

Les fonctions intrinsèques sont spécifiques à chaque compilateur. Ce ne sont pas des fonctions que vous pouvez retrouver dans une bibliothèque. Le compilateur génère une séquence spécifique de code machine lorsqu'il rencontre la fonction intrinsèque. Le plus souvent, il s'agit d'une pseudo fonction qui correspond à une instruction d'un CPU particulier.

Par exemple, il n'existe pas d'opérateur de décalage cyclique dans les langages C/C++. La plupart des CPUs supportent cependant des instructions de ce type. Pour faciliter la vie des programmeurs, le compilateur MSVC propose de telles pseudo fonctions *_rotl()* and *_rotr()*[6] qui sont directement traduites par le compilateur vers les instructions x86 ROL/ROR.

Les fonctions intrinsèques qui permettent de générer des instructions SSE en sont un autre exemple.

La liste complète des fonctions intrinsèques proposées par le compilateur MSVC figurent dans le MSDN.

[6] MSDN

11.5 Anomalies des compilateurs

11.5.1 Oracle RDBMS 11.2 et Intel C++ 10.1

Le compilateur Intel C++ 10.1, qui a été utilisé pour la compilation de Oracle RDBMS 11.2 pour Linux86, émettait parfois deux instructions JZ successives, sans que la seconde instruction soit jamais référencée. Elle était donc inutile.

Listing 11.1 : kdli.o from libserver11.a

```
.text:08114CF1                   loc_8114CF1: ;
    CODE XREF: __PGOSF539_kdlimemSer+89A
.text:08114CF1                                ; __PGOSF539_kdlimemSer+3994
.text:08114CF1 8B 45 08              mov     eax, [ebp+arg_0]
.text:08114CF4 0F B6 50 14           movzx   edx, byte ptr [eax+14h]
.text:08114CF8 F6 C2 01              test    dl, 1
.text:08114CFB 0F 85 17 08 00 00     jnz     loc_8115518
.text:08114D01 85 C9                 test    ecx, ecx
.text:08114D03 0F 84 8A 00 00 00     jz      loc_8114D93
.text:08114D09 0F 84 09 08 00 00     jz      loc_8115518
.text:08114D0F 8B 53 08              mov     edx, [ebx+8]
.text:08114D12 89 55 FC              mov     [ebp+var_4], edx
.text:08114D15 31 C0                 xor     eax, eax
.text:08114D17 89 45 F4              mov     [ebp+var_C], eax
.text:08114D1A 50                    push    eax
.text:08114D1B 52                    push    edx
.text:08114D1C E8 03 54 00 00        call    len2nbytes
.text:08114D21 83 C4 08              add     esp, 8
```

Listing 11.2 : from the same code

```
.text:0811A2A5                   loc_811A2A5: ; CODE XREF: kdliSerLengths+11C
.text:0811A2A5                                ; kdliSerLengths+1C1
.text:0811A2A5 8B 7D 08              mov     edi, [ebp+arg_0]
.text:0811A2A8 8B 7F 10              mov     edi, [edi+10h]
.text:0811A2AB 0F B6 57 14           movzx   edx, byte ptr [edi+14h]
.text:0811A2AF F6 C2 01              test    dl, 1
.text:0811A2B2 75 3E                 jnz     short loc_811A2F2
.text:0811A2B4 83 E0 01              and     eax, 1
.text:0811A2B7 74 1F                 jz      short loc_811A2D8
.text:0811A2B9 74 37                 jz      short loc_811A2F2
.text:0811A2BB 6A 00                 push    0
.text:0811A2BD FF 71 08              push    dword ptr [ecx+8]
.text:0811A2C0 E8 5F FE FF FF        call    len2nbytes
```

Il s'agit probablement d'un bug du générateur de code du compilateur qui ne fut pas découvert durant les tests de celui-ci car le code produit fonctionnait conformément aux résultats attendus.

Un autre exemple tiré d'Oracle RDBMS 11.1.0.6.0 pour win32.

```
.text:0051FBF8 85 C0                             test    eax, eax
.text:0051FBFA 0F 84 8F 00 00 00                 jz      loc_51FC8F
.text:0051FC00 74 1D                             jz      short loc_51FC1F
```

11.5.2 MSVC 6.0

Je viens juste de trouver celui-ci dans un vieux fragment de code :

```
                fabs
                fild    [esp+50h+var_34]
                fabs
                fxch    st(1) ; première instruction
                fxch    st(1) ; seconde instruction
                faddp   st(1), st
                fcomp   [esp+50h+var_3C]
                fnstsw  ax
                test    ah, 41h
                jz      short loc_100040B7
```

La première instruction FXCH intervertit les valeurs de ST(0) et ST(1). La seconde effectue la même opération. Combinées, elles ne produisent donc aucun effet. Cet extrait provient d'un programme qui utilise la bibliothèque MFC42.dll, il a donc dû être compilé avec MSVC 6.0 ou 5.0 ou peut-être même MSVC 4.2 qui date des années 90.

Cette paire d'instructions ne produit aucun effet, ce qui expliquerait qu'elle n'ait pas été détectée lors des tests du compilateur MSVC. Ou bien j'ai loupé quelque chose ...

11.5.3 ftol2() dans MSVC 2012

Je viens de trouver ceci dans la fonction ftol2() de la bibliothèque C/C++ standard (routine de conversion float-to-long) de Microsoft Visual Studio 2012.

```
                public __ftol2
__ftol2         proc near
                push    ebp
                mov     ebp, esp
                sub     esp, 20h
                and     esp, 0FFFFFFF0h
                fld     st
                fst     dword ptr [esp+18h]
                fistp   qword ptr [esp+10h]
                fild    qword ptr [esp+10h]
                mov     edx, [esp+18h]
                mov     eax, [esp+10h]
                test    eax, eax
                jz      short integer_QnaN_or_zero

arg_is_not_integer_QnaN:
                fsubp   st(1), st
                test    edx, edx
                jns     short positive
                fstp    dword ptr [esp]
                mov     ecx, [esp]
                xor     ecx, 80000000h
                add     ecx, 7FFFFFFFh
```

```
                adc     eax, 0
                mov     edx, [esp+14h]
                adc     edx, 0
                jmp     short localexit

positive:
                fstp    dword ptr [esp]
                mov     ecx, [esp]
                add     ecx, 7FFFFFFFh
                sbb     eax, 0
                mov     edx, [esp+14h]
                sbb     edx, 0
                jmp     short localexit

integer_QnaN_or_zero:
                mov     edx, [esp+14h]
                test    edx, 7FFFFFFFh
                jnz     short arg_is_not_integer_QnaN
                fstp    dword ptr [esp+18h] ; first
                fstp    dword ptr [esp+18h] ; second

localexit:
                leave
                retn
__ftol2         endp
```

Notez les deux FSTP-s (stocker un float avec pop) identiques à la fin. D'abord, j'ai cru qu'il s'agissait d'une anomalie du compilateur (je collectionne de tels cas tout comme certains collectionnent les papillons), mais il semble qu'il s'agisse d'un morceau d'assembleur écrit à la main, dans msvcrt.lib il y a un fichier objet avec cette fonction dedans et on peut y trouver cette chaîne:
`f:\dd\vctools\crt_bld\SELF_X86\crt\prebuild\tran\i386\ftol2.asm` — qui est sans doute un chemin vers le fichier sur l'ordinateur du développeur où msvcrt.lib a été généré.

Donc, bogue, typo induite par l'éditeur de texte ou fonctionnalité?

11.5.4 Résumé

Des anomalies constatées dans d'autres compilateurs figurent également dans ce livre: 1.28.2 on page 405, 3.10.3 on page 630, 3.18.7 on page 684, 1.26.7 on page 386, 1.18.4 on page 194, 1.28.5 on page 428.

Ces cas sont exposés dans ce livre afin de démontrer que ces compilateurs comportent leurs propres erreurs et qu'il convient de ne pas toujours se torturer le cerveau en tentant de comprendre pourquoi le compilateur a généré un code aussi étrange.

11.6 Itanium

Bien qu'elle n'ai pas réussi à percer, l'architecture Intel Itanium (IA64) est très intéressante.

Là où les CPUs OOE réarrangent les instructions afin de les exécuter en parallèle, l'architecture EPIC[7] a constitué une tentative pour déléguer cette décision au compilateur.

Les compilateurs en question étaient évidemment particulièrement complexes.

Voici un exemple de code pour l'architecture IA64 qui implémente un algorithme de chiffrage simple du noyau Linux:

Listing 11.3 : Linux kernel 3.2.0.4

```
#define TEA_ROUNDS              32
#define TEA_DELTA               0x9e3779b9

static void tea_encrypt(struct crypto_tfm *tfm, u8 *dst, const u8 *src)
{
        u32 y, z, n, sum = 0;
        u32 k0, k1, k2, k3;
        struct tea_ctx *ctx = crypto_tfm_ctx(tfm);
        const __le32 *in = (const __le32 *)src;
        __le32 *out = (__le32 *)dst;

        y = le32_to_cpu(in[0]);
        z = le32_to_cpu(in[1]);

        k0 = ctx->KEY[0];
        k1 = ctx->KEY[1];
        k2 = ctx->KEY[2];
        k3 = ctx->KEY[3];

        n = TEA_ROUNDS;

        while (n-- > 0) {
                sum += TEA_DELTA;
                y += ((z << 4) + k0) ^ (z + sum) ^ ((z >> 5) + k1);
                z += ((y << 4) + k2) ^ (y + sum) ^ ((y >> 5) + k3);
        }

        out[0] = cpu_to_le32(y);
        out[1] = cpu_to_le32(z);
}
```

Et voici maintenant le résultat de la compilation:

Listing 11.4 : Linux Kernel 3.2.0.4 pour Itanium 2 (McKinley)

```
0090|                   tea_encrypt:
0090|08 80 80 41 00 21      adds r16 = 96, r32              // ptr to ctx->⤦
    ↳ KEY[2]
```

[7] Explicitly Parallel Instruction Computing

```
0096|80 C0 82 00 42 00      adds r8 = 88, r32              // ptr to ctx->↙
    ↳ KEY[0]
009C|00 00 04 00            nop.i 0
00A0|09 18 70 41 00 21      adds r3 = 92, r32              // ptr to ctx->↙
    ↳ KEY[1]
00A6|F0 20 88 20 28 00      ld4 r15 = [r34], 4             // load z
00AC|44 06 01 84            adds r32 = 100, r32;;          // ptr to ctx->↙
    ↳ KEY[3]
00B0|08 98 00 20 10 10      ld4 r19 = [r16]                // r19=k2
00B6|00 01 00 00 42 40      mov r16 = r0                   // r0 always ↙
    ↳ contain zero
00BC|00 08 CA 00            mov.i r2 = ar.lc               // save lc ↙
    ↳ register
00C0|05 70 00 44 10 10
     9E FF FF FF 7F 20      ld4 r14 = [r34]                // load y
00CC|92 F3 CE 6B            movl r17 = 0xFFFFFFFF9E3779B9;; // TEA_DELTA
00D0|08 00 00 00 01 00      nop.m 0
00D6|50 01 20 20 20 00      ld4 r21 = [r8]                 // r21=k0
00DC|F0 09 2A 00            mov.i ar.lc = 31               // TEA_ROUNDS ↙
    ↳ is 32
00E0|0A A0 00 06 10 10      ld4 r20 = [r3];;               // r20=k1
00E6|20 01 80 20 20 00      ld4 r18 = [r32]                // r18=k3
00EC|00 00 04 00            nop.i 0
00F0|
00F0|                                  loc_F0:
00F0|09 80 40 22 00 20      add r16 = r16, r17             // r16=sum, r17↙
    ↳ =TEA_DELTA
00F6|D0 71 54 26 40 80      shladd r29 = r14, 4, r21       // r14=y, r21=↙
    ↳ k0
00FC|A3 70 68 52            extr.u r28 = r14, 5, 27;;
0100|03 F0 40 1C 00 20      add r30 = r16, r14
0106|B0 E1 50 00 40 40      add r27 = r28, r20;;           // r20=k1
010C|D3 F1 3C 80            xor r26 = r29, r30;;
0110|0B C8 6C 34 0F 20      xor r25 = r27, r26;;
0116|F0 78 64 00 40 00      add r15 = r15, r25             // r15=z
011C|00 00 04 00            nop.i 0;;
0120|00 00 00 00 01 00      nop.m 0
0126|80 51 3C 34 29 60      extr.u r24 = r15, 5, 27
012C|F1 98 4C 80            shladd r11 = r15, 4, r19       // r19=k2
0130|0B B8 3C 20 00 20      add r23 = r15, r16;;
0136|A0 C0 48 00 40 00      add r10 = r24, r18             // r18=k3
013C|00 00 04 00            nop.i 0;;
0140|0B 48 28 16 0F 20      xor r9 = r10, r11;;
0146|60 B9 24 1E 40 00      xor r22 = r23, r9
014C|00 00 04 00            nop.i 0;;
0150|11 00 00 00 01 00      nop.m 0
0156|E0 70 58 00 40 A0      add r14 = r14, r22
015C|A0 FF FF 48            br.cloop.sptk.few loc_F0;;
0160|09 20 3C 42 90 15      st4 [r33] = r15, 4             // store z
0166|00 00 00 02 00 00      nop.m 0
016C|20 08 AA 00            mov.i ar.lc = r2;;             // restore lc ↙
    ↳ register
0170|11 00 38 42 90 11      st4 [r33] = r14                // store y
```

```
0176|00 00 00 02 00 80      nop.i 0
017C|08 00 84 00            br.ret.sptk.many b0;;
```

Nous constatons tout d'abord que toutes les instructions IA64 sont regroupées par 3.

Chaque groupe représente 16 octets (128 bits) et se décompose en une catégorie de code sur 5 bits puis 3 instructions de 41 bits chacune.

Dans IDA les groupes apparaissent sous la forme 6+6+4 octets —le motif est facilement reconnaissable.

En règle générale les trois instructions d'un groupe s'exécutent en parallèle, sauf si l'une d'elles est associée à un «stop bit».

Il est probable que les ingénieurs d'Intel et de HP ont collecté des statistiques qui leur ont permis d'identifier les motifs les plus fréquents. Ils auraient alors décidé d'introduire une notion de type de groupe (AKA «templates»). Le type du groupe définit la catégorie des instructions qu'il contient. Ces catégories sont au nombre de 12.

Par exemple, un groupe de type 0 représente `MII`. Ceci signifie que la première instruction est une lecture ou écriture en mémoire (M), la seconde et la troisième sont des manipulations d'entiers (I).

Un autre exemple est le groupe de type 0x1d: `MFB`. La première instruction est la aussi de type mémoire (M), la seconde manipule un nombre flottant (F instruction FPU), et la dernière est un branchement (B).

Lorsque le compilateur ne parvient pas à sélectionner une instruction à inclure dans le groupe en cours de construction, il utilise une instruction de type NOP. Il existe donc des instructions `nop.i` pour remplacer ce qui devrait être une manipulation d'entier. De même un `nop.m` est utilisé pour combler un trou là où une instruction de type mémoire devrait se trouver.

Lorsque le programme est directement rédigé en assembleur, les instructions NOPs sont ajoutées de manière automatique.

Et ce n'est pas tout. Les groupes font eux-mêmes l'objet de regroupements.

Chaque instruction peut être marquée avec un «stop bit». Le processeur exécute en parallèle toutes les instructions, jusqu'à ce qu'il rencontre un «stop bit».

En pratique, les processeurs Itanium 2 peuvent exécuter jusqu'à deux groupes simultanément, soit un total de 6 instructions en parallèle.

Il faut évidemment que les instructions exécutées en parallèle, n'aient pas d'effet de bord entre elles. Dans le cas contraire, le résultat n'est pas défini. Le compilateur doit respecter cette contrainte ainsi que le nombre maximum de groupes simultanés du processeur cible en plaçant les «stop bit» au bon endroit.

En langage d'assemblage, les bits d'arrêt sont identifiés par deux point-virgule situés après l'instruction.

Ainsi dans notre exemple les instructions [90-ac] peuvent être exécutées simultanément. Le prochain groupe est [b0-cc].

Nous observons également un bit d'arrêt en 10c. L'instruction suivante comporte elle aussi un bit d'arrêt.

Ces deux instructions doivent donc être exécutées isolément des autres, (comme dans une architecture CISC).

En effet, l'instruction en 110 utilise le résultat produit par l'instruction précédente (valeur du registre r26). Les deux instructions ne peuvent s'exécuter simultanément.

Il semble que le compilateur n'ai pas trouvé de meilleure manière de paralléliser les instructions, ou en d'autres termes, de plus charger la CPU. Les bits d'arrêt sont donc en trop grand nombre.

La rédaction manuelle de code en assembleur est une tâche pénible. Le programmeur doit effectuer lui-même les regroupements d'instructions.

Bien sûr, il peut ajouter un bit d'arrêt à chaque instruction mais cela dégrade les performances telles qu'elles ont été pensée pour l'Itanium.

Les codes source du noyau Linux contiennent un exemple intéressant d'un code assembleur produit manuellement pour IA64 :

`http://lxr.free-electrons.com/source/arch/ia64/lib/`.

On trouvera une introduction à l'assembleur Itanium dans : [Mike Burrell, *Writing Efficient Itanium 2 Assembly Code* (2010)][8], [papasutra of haquebright, *WRITING SHELLCODE FOR IA-64* (2001)][9].

Deux autres caractéristiques très intéressantes d'Itanium sont l'*exécution spéculative* et le bit NaT («not a thing») qui ressemble un peu aux nombres NaN : MSDN.

11.7 Modèle de mémoire du 8086

Lorsque l'on a à faire avec des programmes 16-bit pour MS-DOS ou Win16 (8.8.3 on page 1105 ou 3.34.5 on page 846), nous voyons que les pointeurs consistent en deux valeurs 16-bit. Que signifient-elles? Eh oui, c'est encore un artefact étrange de MS-DOS et du 8086.

Le 8086/8088 était un CPU 16-bit, mais était capable d'accèder à des adresses mémoire sur 20-bit (il était donc capable d'accèder 1MB de mémoire externe).

L'espace de la mémoire externe était divisé entre la RAM (max 640KB), la ROM, la fenêtre pour la mémoire vidéo, les cartes EMS, etc.

Rappelons que le 8086/8088 était en fait un descendant du CPU 8-bit 8080.

Le 8080 avait un espace mémoire de 16-bit, i.e., il pouvait seulement adresser 64KB.

Et probablement pour une raison de portage de vieux logiciels[10], le 8086 peut supporter plusieurs fenêtres de 64KB simultanément, situées dans l'espace d'adresse de 1MB.

[8]Aussi disponible en `http://yurichev.com/mirrors/RE/itanium.pdf`
[9]Aussi disponible en `http://phrack.org/issues/57/5.html`
[10]Je ne suis pas sûr à 100% de ceci

C'est une sorte de virtualisation de niveau jouet.

Tous les registres 8086 sont 16-bit, donc pour adresser plus, des registres spéciaux de segment (CS, DS, ES, SS) ont été ajoutés.

Chaque pointeur de 20-bit est calculé en utilisant les valeurs d'une paire de registres, de segment et d'adresse (p.e. DS:BX) comme suit:

$$real_address = (segment_register \ll 4) + address_register$$

Par exemple, la fenêtre de RAM graphique (EGA[11], VGA[12]) sur les vieux compatibles IBM-PC a une taille de 64KB.

Pour y accèder, une valeur de 0xA000 doit être stockée dans l'un des registres de segments, p.e. dans DS.

Ainsi DS:0 adresse le premier octet de la RAM vidéo et DS:0xFFFF — le dernier octet de RAM.

L'adresse réelle sur le bus d'adresse de 20-bit, toutefois, sera dans l'intervalle 0xA0000 à 0xAFFFF.

Le programme peut contenir des adresses codées en dur comme 0x1234, mais l'OS peut avoir besoin de le charger à une adresse arbitraire, donc il recalcule les valeurs du registre de segment de façon à ce que le programme n'ait pas à se soucier de l'endroit où il est placé dans la RAM.

Donc, tout pointeur dans le vieil environnement MS-DOS consistait en fait en un segment d'adresse et une adresse dans ce segment, i.e., deux valeurs 16-bit. 20-bit étaient suffisants pour cela, cependant nous devions recalculer les adresses très souvent: passer plus d'informations par la pile semblait un meilleur rapport espace/facilité.

À propos, à cause de tout cela, il n'était pas possible d'allouer un bloc de mémoire plus large que 64KB.

Les registres de segment furent réutilisés sur les 80286 comme sélecteurs, servant a une fonction différente.

Lorsque les CPU 80386 et les ordinateurs avec plus de RAM ont été introduits, MS-DOS était encore populaire, donc des extensions pour DOS ont émergés: ils étaient en fait une étape vers un OS «sérieux», basculant le CPU en mode protégé et fournissant des APIs mémoire bien meilleures pour les programmes qui devaient toujours fonctionner sous MS-DOS.

Des examples très populaires incluent DOS/4GW (le jeux vidéo DOOM a été compilé pour lui), Phar Lap, PMODE.

À propos, la même manière d'adresser la mémoire était utilisée dans la série 16-bit de Windows 3.x, avant Win32.

[11]Enhanced Graphics Adapter
[12]Video Graphics Array

11.8 Réordonnancement des blocs élémentaires

11.8.1 Optimisation guidée par profil

Cette méthode d'optimisation déplace certains basic blocks vers d'autres sections du fichier binaire exécutable.

Il est évident que certaines parties d'une fonction sont exécutées plus fréquemment que d'autres (ex: le corps d'une boucle) et d'autres moins souvent (ex: gestionnaire d'erreur, gestionnaires d'exception).

Le compilateur ajoute dans le code exécutable des instructions d'instrumentation. Le développeur exécute ensuite un nombre important de tests, ce qui permet de collecter des statistiques.

A l'aider de ces dernières, le compilateur prépare le fichier exécutable final en déplacant les fragments de code les moins exécutés vers une autre section.

Tous les fragments de code les plus souvent exécutés sont ainsi regroupés, ce qui constitue un facteur important pour la rapidité d'exécution et l'utilisation du cache.

Voici un exemple de code Oracle RDBMS produit par le compilateur Intel C++:

Listing 11.5 : orageneric11.dll (win32)

```
                public _skgfsync
_skgfsync       proc near

; address 0x6030D86A

                db      66h
                nop
                push    ebp
                mov     ebp, esp
                mov     edx, [ebp+0Ch]
                test    edx, edx
                jz      short loc_6030D884
                mov     eax, [edx+30h]
                test    eax, 400h
                jnz     __VInfreq__skgfsync  ; write to log
continue:
                mov     eax, [ebp+8]
                mov     edx, [ebp+10h]
                mov     dword ptr [eax], 0
                lea     eax, [edx+0Fh]
                and     eax, 0FFFFFFFCh
                mov     ecx, [eax]
                cmp     ecx, 45726963h
                jnz     error                ; exit with error
                mov     esp, ebp
                pop     ebp
                retn
_skgfsync       endp

...
```

```
; address 0x60B953F0

__VInfreq__skgfsync:
                mov     eax, [edx]
                test    eax, eax
                jz      continue
                mov     ecx, [ebp+10h]
                push    ecx
                mov     ecx, [ebp+8]
                push    edx
                push    ecx
                push    offset ... ;
    "skgfsync(se=0x%x, ctx=0x%x, iov=0x%x)\n"
                push    dword ptr [edx+4]
                call    dword ptr [eax] ; write to log
                add     esp, 14h
                jmp     continue

error:
                mov     edx, [ebp+8]
                mov     dword ptr [edx], 69AAh ; 27050 "function called with
    invalid FIB/IOV structure"
                mov     eax, [eax]
                mov     [edx+4], eax
                mov     dword ptr [edx+8], 0FA4h ; 4004
                mov     esp, ebp
                pop     ebp
                retn
; END OF FUNCTION CHUNK FOR _skgfsync
```

La distance entre ces deux fragments de code avoisine les 9 Mo.

Tous les fragments de code rarement exécutés sont regroupés à la fin de la section de code de la DLL.

La partie de la fonction qui a été déplacée était marquée par le compilateur Intel C++ avec le préfixe `VInfreq`.

Nous voyons donc qu'une partie de la fonction qui écrit dans un fichier journal (probablement à la suite d'une erreur ou d'un avertissement) n'a sans doute pas été exécuté très souvent durant les tests effectués par les développeurs Oracle lors de la collecte des statistiques. Il n'est même pas dit qu'elle ait jamais été exécutée.

Le bloc élémentaire qui écrit dans le journal s'achève par un retour à la partie « hot » de la fonction.

Un autre bloc élémentaire « infrequent » est celui qui retourne le code erreur 27050.

Pour ce qui est des fichiers Linux au format ELF, le compilateur Intel C++ déplace tous les fragments de code rarement exécutés vers une section séparée nommée `text.unlikely`. Les fragments les plus souvent exécutés sont quant à eux regroupés dans la section `text.hot`.

Cette information peut aider le rétro ingénieur à distinguer la partie principale d'une

fonction des parties qui assurent la gestion d'erreurs.

11.9 Mon expérience avec Hex-Rays 2.2.0

11.9.1 Bugs

Il y a plusieurs bugs.

Tout d'abord, Hex-Rays est perdu lorsque des instructions FPU sont mélangées (par le générateur de code du compilateur) avec des autres.

Par exemple, ceci:

```
f               proc    near

                lea     eax, [esp+4]
                fild    dword ptr [eax]
                lea     eax, [esp+8]
                fild    dword ptr [eax]
                fabs
                fcompp
                fnstsw  ax
                test    ah, 1
                jz      l01

                mov     eax, 1
                retn
l01:
                mov     eax, 2
                retn

f               endp
```

…sera correctement décompilé en:

```
signed int __cdecl f(signed int a1, signed int a2)
{
  signed int result; // eax@2

  if ( fabs((double)a2) >= (double)a1 )
    result = 2;
  else
    result = 1;
  return result;
}
```

Mais commentons une des instructions à la fin:

```
...
l01:
                ;mov    eax, 2
                retn
...
```

...nous obtenons ce bug évident:

```
void __cdecl f(char a1, char a2)
{
  fabs((double)a2);
}
```

Ceci est un autre bug:

```
extrn f1:dword
extrn f2:dword

f               proc    near

                fld     dword ptr [esp+4]
                fadd    dword ptr [esp+8]
                fst     dword ptr [esp+12]
                fcomp   ds:const_100
                fld     dword ptr [esp+16]      ; commenter cette ↙
    ↳ instruction et ça sera OK
                fnstsw  ax
                test    ah, 1

                jnz     short l01

                call    f1
                retn
l01:
                call    f2
                retn

f               endp

...

const_100       dd 42C80000h            ; 100.0
```

Résultat:

```
int __cdecl f(float a1, float a2, float a3, float a4)
{
  double v5; // st7@1
  char v6; // c0@1
  int result; // eax@2

  v5 = a4;
  if ( v6 )
    result = f2(v5);
  else
    result = f1(v5);
  return result;
}
```

La variable v6 a un type char et si vous essayez de compiler ce code, le compilateur vous avertira à propos de l'utilisation de variable avant son initialisation.

Un autre bug: l'instruction FPATAN est correctement décompilée en atan2(), mais les arguments sont échangés.

11.9.2 Particularités bizarres

Hex-Rays converti trop souvent des int 32-bit en 64-bit. Voici un exemple:

```
f               proc    near

                mov     eax, [esp+4]
                cdq
                xor     eax, edx
                sub     eax, edx
                ; EAX=abs(a1)

                sub     eax, [esp+8]
                ; EAX=EAX-a2

                ; EAX à ce point est converti en 64-bit (RAX)

                cdq
                xor     eax, edx
                sub     eax, edx
                ; EAX=abs(abs(a1)-a2)

                retn

f               endp
```

Résultat:

```
int __cdecl f(int a1, int a2)
{
  __int64 v2; // rax@1

  v2 = abs(a1) - a2;
  return (HIDWORD(v2) ^ v2) - HIDWORD(v2);
}
```

Peut-être est-ce le résultat de l'instruction CDQ? Je ne suis pas sûr. Quoiqu'il en soit, à chaque fois que vous voyez le type __int64 dans du code 32-bit, soyez attentif.

Ceci est aussi bizarre:

```
f               proc    near

                mov     esi, [esp+4]

                lea     ebx, [esi+10h]
                cmp     esi, ebx
                jge     short l00
```

```
                cmp     esi, 1000
                jg      short l00

                mov     eax, 2
                retn

l00:
                mov     eax, 1
                retn

f               endp
```

Résultat:

```
signed int __cdecl f(signed int a1)
{
  signed int result; // eax@3

  if ( __OFSUB__(a1, a1 + 16) ^ 1 && a1 <= 1000 )
    result = 2;
  else
    result = 1;
  return result;
}
```

Le code est correct, mais il requiert une intervention manuelle.

Parfois, Hex-Rays ne remplace pas le code de la division par la multiplication:

```
f               proc    near

                mov     eax, [esp+4]
                mov     edx, 2AAAAAABh
                imul    edx
                mov     eax, edx

                retn

f               endp
```

Résultat:

```
int __cdecl f(int a1)
{
  return (unsigned __int64)(715827883i64 * a1) >> 32;
}
```

Ceci peut être remplacé manuellement.

Beaucoup de ces particularités peuvent être résolues en ré-arrangeant les instructions, recompilant le code assembleur et en le renvoyant dans Hex-Rays.

11.9.3 Silence

```
extrn some_func:dword

f               proc    near

                mov     ecx, [esp+4]
                mov     eax, [esp+8]
                push    eax
                call    some_func
                add     esp, 4

                ; use ECX
                mov     eax, ecx

                retn

f               endp
```

Résultat:

```
int __cdecl f(int a1, int a2)
{
  int v2; // ecx@1

  some_func(a2);
  return v2;
}
```

La variable v2 (de ECX) est perdue …Oui, ce code est incorrect (la valeur de ECX n'est pas sauvée lors de l'appel d'une autre fonction), mais il serait bon que Hex-Rays donne un warning.

Un autre:

```
extrn some_func:dword

f               proc    near

                call    some_func
                jnz     l01

                mov     eax, 1
                retn
l01:
                mov     eax, 2
                retn

f               endp
```

Résultat:

```
signed int f()
{
```

```
  char v0; // zf@1
  signed int result; // eax@2

  some_func();
  if ( v0 )
    result = 1;
  else
    result = 2;
  return result;
}
```

Encore une fois, un warning serait utile.

En tout cas, à chaque fois que vous voyez une variable de type char, ou une variable qui est utilisée sans initialisation, c'est un signe clair que quelque chose s'est mal passé et nécessite une intervention manuelle.

11.9.4 Virgule

La virgule en C/C++ a mauvaise presse, car elle peut conduire à du code confus.

Quiz rapide, que renvoie cette fonction C/C++ ?

```
int f()
{
        return 1, 2;
};
```

C'est 2: lorsque le compilateur rencontre une expression avec des virgules, il génère du code qui exécute toutes les sous-expressions, et renvoie la valeur de la dernière.

J'ai vu quelque chose comme ça dans du code en production:

```
if (cond)
        return global_var=123, 456; // 456 is returned
else
        return global_var=789, 321; // 321 is returned
```

Il semble que le programmeur voulait rendre le code plus court sans parenthèses supplémentaires. Autrement dit, la virgule permet de grouper plusieurs expressions en une seule, sans déclaration/bloc de code dans des parenthèses.

La virgule en C/C++ est proche du begin en Scheme/Racket: https://docs.racket-lang.org/guide/begin.html.

Peut-être que le seul usage largement accepté de la virgule est dans les déclarations for() :

```
char *s="hello, world";
for(int i=0; *s; s++, i++);
; i = string lenght
```

À la fois s++ et i++ sont exécutés à chaque itération de la boucle.

Plus d'information:
http://stackoverflow.com/questions/52550/what-does-the-comma-operator-do-in-c.

J'ai écrit tout ceci car Hex-Rays produit (au moins dans mon cas) du code qui est riche tant en virgules qu'en expression raccourcies: Par exemple, ceci est une sortie réelle de Hex-Rays:

```
 if ( a >= b || (c = a, (d[a] - e) >> 2 > f) )
    {
        ...
```

Ceci est correct, compile et fonctionne, et Dieu puisse vous aider à la comprendre. La voici récrite:

```
if (cond1 || (comma_expr, cond2))
{
        ...
```

Le raccourci est effectif ici: d'abord cond1 est testé, si c'est `true`, le corps du `if()` est exécuté, le reste de l'expression du `if()` est complètement ignoré. Si cond1 est `false`, comma_expr est exécuté (dans l'exemple précédent, a est copié dans c), puis cond2 est testée. Si cond2 est `true`, le corps du `if()` est exécuté, ou pas. Autrement dit, le corps du `if()` est exécuté si cond1 est `true` ou si cond2 est `true`, mais si ce dernier est `true`, comma_expr est aussi exécutée.

Maintenant, vous pouvez voir pourquoi la virgule est si célèbre.

Un mot sur les raccourcis. Une idée fausse répandue de débutant est que les sous-conditions sont testées dans un ordre indéterminé, ce qui n'est pas vrai. Dans l'expression `a | b | c`, a, b et c sont évalués dans un ordre indéterminé, donc c'est pourquoi || a été ajouté à C/C++, pour appliquer des raccourcis explicitement.

11.9.5 Types de donnée

Les types de donnée sont un problème pour les décompilateurs.

Hex-Rays peut ne pas voir les tableaux dans la pile locale, si ils n'ont pas été déterminés avant la décompilation. Même histoire avec les tableaux globaux.

Un autre problème se pose avec les fonctions trop grosses, où un slot unique dans la pile locale peut être utilisé par plusieurs variables durant l'exécution de la fonction. Ce n'est pas un cas rare que lorsqu'un slot est utilisé pour une variable `int`, puis un pointeur, puis une variable `float`. Hex-Rays le décompile correctement: il créé une variable avec le même type, puis la caste sur un autre type dans diverses parties de la fonction. J'ai résolu ce problème en découpant les grosses fonctions en plusieurs plus petites. Met les variables locales comme des globales, etc., etc. Et n'oubliez pas les tests.

11.9.6 Expressions longues et confuses

Parfois, lors de la ré-écriture, vous pouvez vous retrouvez avec des expressions longues et difficiles à comprendre dans des constructions `if()` comme:

```
if ((! (v38 && v30 <= 5 && v27 != -1)) && ((! (v38 && v30 <= 5) && v27 != ↙
    ↳ -1) || (v24 >= 5 || v26)) && v25)
{
...
}
```

Wolfram Mathematica peut minimiser certaines d'entre elles, en utilisant la fonction `BooleanMinimize[]` :

```
In[1]:= BooleanMinimize[(! (v38 && v30 <= 5 && v27 != -1)) && v38 && v30 <=↙
    ↳  5 && v25 == 0]

Out[1]:= v38 && v25 == 0 && v27 == -1 && v30 <= 5
```

Il y a encore une meilleure voie, pour trouver les sous-expressions communes:

```
In[2]:= Experimental`OptimizeExpression[(! (v38 && v30 <= 5 &&
      v27 != -1)) && ((! (v38 && v30 <= 5) &&
      v27 != -1) || (v24 >= 5 || v26)) && v25]

Out[2]= Experimental`OptimizedExpression[
 Block[{Compile`$1, Compile`$2}, Compile`$1 = v30 <= 5;
  Compile`$2 =
   v27 != -1; ! (v38 && Compile`$1 &&
      Compile`$2) && ((! (v38 && Compile`$1) && Compile`$2) ||
     v24 >= 5 || v26) && v25]]
```

Mathematica ajoute deux nouvelles variables: `Compile`$1` et `Compile`$2`, qui vont être ré-utilisées plusieurs fois dans l'expression. Donc, nous pouvons ajouter deux variables supplémentaires.

11.9.7 Loi de De Morgan et décompilation

Parfois l'optimiseur du compilateur peut utiliser la loi de De Morgan pour rendre le code plus court/rapide.

Par exemple, ceci:

```
void f(int a, int b, int c, int d)
{
        if (a>0 && b>0)
                printf ("both a and b are positive\n");
        else if (c>0 && d>0)
                printf ("both c and d are positive\n");
        else
                printf ("something else\n");
};
```

…ça semble assez anodin, lorsque c'est compilé avec GCC 5.4.0 x64 avec optimisation:

```
; int __fastcall f(int a, int b, int c, int d)
                public f
f               proc near
                test    edi, edi
                jle     short loc_8
                test    esi, esi
                jg      short loc_30

loc_8:
                test    edx, edx
                jle     short loc_20
                test    ecx, ecx
                jle     short loc_20
                mov     edi, offset s   ; "both c and d are positive"
                jmp     puts

loc_20:
                mov     edi, offset aSomethingElse ; "something else"
                jmp     puts

loc_30:
                mov     edi, offset aAAndBPositive ; "both a and b are
    positive"

loc_35:
                jmp     puts
f               endp
```

…ça semble donc anodin, mais Hex-Rays 2.2.0 n'arrive pas vraiment à détecter qu'en fait des opérations double AND ont été utilisées dans le code source:

```
int __fastcall f(int a, int b, int c, int d)
{
  int result;

  if ( a > 0 && b > 0 )
  {
    result = puts("both a and b are positive");
  }
  else if ( c <= 0 || d <= 0 )
  {
    result = puts("something else");
  }
  else
  {
    result = puts("both c and d are positive");
  }
  return result;
}
```

L'expression `c <= 0 || d <= 0` est la contraposée de `c>0 && d>0` puisque $\overline{A \cup B} = \overline{A} \cap \overline{B}$ et $\overline{A \cap B} = \overline{A} \cup \overline{B}$, Autrement dit, `!(cond1 || cond2) == !cond1 && !cond2` et

`!(cond1 && cond2) == !cond1 || !cond2`.

Ça vaut la peine de garder ces règles à l'esprit, puisque ces optimisations du compilateur sont utilisées massivement presque partout.

C'est parfois une bonne idée d'inverser une condition, afin de mieux comprendre le code. Ceci est un morceau de code réel décompilé par Hex-Rays:

```
        for (int i=0; i<12; i++)
        {
                if (v1[i-12] != 0.0 || v1[i] != 0.0)
                {
                        v108=min(v108, (float)v0[i*24 -2]);
                        v113=max(v113, (float)v0[i*24]);
                };
        }
```

…qui peut être récrit comme:

```
        for (int i=0; i<12; i++)
        {
                if (v1[i-12] == 0.0 && v1[i] == 0.0)
                        continue;

                v108=min(v108, (float)v0[i*24 -2]);
                v113=max(v113, (float)v0[i*24]);
        }
```

Lequel est le meilleur? Je ne sais pas encore, mais pour une meilleure compréhension, c'est bien de regarder les deux.

11.9.8 Mon plan

- Séparer les grosses fonctions (et ne pas oublier de tester). Parfois c'est utile de créer des nouvelles fonctions à partir des corps de boucles.
- Vérifier/tester le type des variables, tableaux, etc.
- Si vous voyez un résultat étrange, une variable `dangling` (qui est utilisée avant son initialisation), essayez d'échanger les instructions manuellement, recompilez et repassez-le à Hex-Rays.

11.9.9 Résumé

Quoiqu'il en soit, la qualité de Hex-Rays 2.2.0 est très, très bonne. Il rend la vie plus facile.

11.10 Complexité cyclomatique

Ce terme est utilisé pour mesurer la complexité d'une fonction. Les fonctions complexes sont souvent un fléau, car elles sont difficiles à maintenir, difficiles à tester, etc.

Il y a plusieurs heuristiques pour la mesurer.

Par exemple, on trouve dans la façon de coder du noyau Linux[13] :

> Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.
>
> ...
>
> Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.
>
> The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's OK to have a longer function.
>
> However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names (you can ask the compiler to in-line them if you think it's performance-critical, and it will probably do a better job of it than you would have done).
>
> Another measure of the function is the number of local variables. They shouldn't exceed 5-10, or you're doing something wrong. Re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confused. You know you're brilliant, but maybe you'd like to understand what you did 2 weeks from now.

Dans le JPL Institutional Coding Standard for the C Programming Language[14] :

> Functions should be no longer than 60 lines of text and define no more than 6 parameters.
>
> A function should not be longer than what can be printed on a single sheet of paper in a standard reference format with one line per statement and one line per declaration. Typically, this means no more than about 60 lines of code per function. Long lists of function parameters similarly compromise code clarity and should be avoided.
>
> Each function should be a logical unit in the code that is understandable and verifiable as a unit. It is much harder to understand a logical unit that spans multiple screens on a computer display or multiple pages when printed. Excessively long functions are often a sign

[13] https://www.kernel.org/doc/html/v4.10/process/coding-style.html
[14] https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf

of poorly structured code.

Revenons maintenant à la complexité cyclomatique.

Sans plonger profondément dans la théorie des graphes: il y a des blocs de base et des liens entre eux. Par exemple, voici comment IDA montre les BB[15]s et les liens (avec des flèches). En appuyant sur la barre d'espace, vous verrez ceci: 1.18 on page 122. Chaque BB est aussi appelé vertex ou nœ dans la théorie des graphes. Chaque lien - arête.

Il y a au moins deux façons courantes de calculer la complexité cyclomatique: 1) arêtes - nœds + 2 2) arêtes - nœds + nombre de sorties (instructions RET)

Dans l'exemple avec IDA ici-dessous, il y a 4 BBs, donc il y a 4 nœds. Mais il y aussi 4 liens et une instruction de retour.

Plus le nombre est grand, plus votre fonction est complexe et les choses vont de mal en pis. Comme vous pouvez le voir, une sortie supplémentaire (instruction return) rend les choses encore pire, tout comme des liens additionnels entre les nœds (un goto additionnel inclus).

J'ai écris un simple script IDAPython (https://beginners.re/paywall/RE4B-source/current-tree//other/cyclomatic/cyclomatic.py) pour la mesurer. Voici le résultat pour le noyau Linux 4.11 (ses fonctions les plus complexes) :

```
1829c0 do_check edges=937 nodes=574 rets=1 E-N+2=365 E-N+rets=364
2effe0 ext4_fill_super edges=862 nodes=568 rets=1 E-N+2=296 E-N+rets=295
5d92e0 wm5110_readable_register edges=661 nodes=369 rets=2 E-N+2=294 E-N+⤦
    ↳ rets=294
277650 do_blockdev_direct_IO edges=771 nodes=507 rets=1 E-N+2=266 E-N+rets⤦
    ↳ =265
10f7c0 load_module edges=711 nodes=465 rets=1 E-N+2=248 E-N+rets=247
787730 dev_ethtool edges=559 nodes=315 rets=1 E-N+2=246 E-N+rets=245
84e440 do_ipv6_setsockopt edges=468 nodes=237 rets=1 E-N+2=233 E-N+rets=232
72c3c0 mmc_init_card edges=593 nodes=365 rets=1 E-N+2=230 E-N+rets=229
...
```

(Liste complète: https://beginners.re/paywall/RE4B-source/current-tree//other/cyclomatic/linux_4.11_sorted.txt)

Voici le code source de certaines d'entre elles: do_check(), ext4_fill_super(), do_blockdev_direct_IO(), do_jit().

Fonctions les plus complexes du fichier ntoskrnl.exe de Windows 7:

```
140569400 sub_140569400 edges=3070 nodes=1889 rets=1 E-N+2=1183 E-N+rets⤦
    ↳ =1182
14007c640 MmAccessFault edges=2256 nodes=1424 rets=1 E-N+2=834 E-N+rets=833
1401a0410 FsRtlMdlReadCompleteDevEx edges=1241 nodes=752 rets=1 E-N+2=491 E⤦
    ↳ -N+rets=490
14008c190 MmProbeAndLockPages edges=983 nodes=623 rets=1 E-N+2=362 E-N+rets⤦
    ↳ =361
```

[15] Basic Block

```
14037fd10 ExpQuerySystemInformation edges=995 nodes=671 rets=1 E-N+2=326 E-
    N+rets=325
140197260 MmProbeAndLockSelectedPages edges=875 nodes=551 rets=1 E-N+2=326
    E-N+rets=325
140362a50 NtSetInformationProcess edges=880 nodes=586 rets=1 E-N+2=296 E-N+
    rets=295
....
```

(Liste complète: https://beginners.re/paywall/RE4B-source/current-tree//other/cyclomatic/win7_ntoskrnl_sorted.txt)

Du point de vue du chasseur de bogues, les fonctions complexes sont plus susceptibles d'avoir des bogues, donc il faut leurs donner une attention particulière.

En lire plus à ce sujet: https://fr.wikipedia.org/wiki/Nombre_cyclomatique, https://en.wikipedia.org/wiki/Cyclomatic_complexity, http://wiki.c2.com/?CyclomaticComplexityMetric.

Mesure de la complexité cyclomatique dans MSVS (C#) : https://blogs.msdn.microsoft.com/zainnab/2011/05/17/code-metrics-cyclomatic-complexity/.

Une paire d'autres scripts Python pour mesurer la complexité dans IDA : http://www.openrce.org/articles/full_view/11, https://github.com/mxmssh/IDAmetrics (incl. other metrics).

Plugin GCC: https://github.com/ephox-gcc-plugins/cyclomatic_complexity.

Chapitre 12

Livres/blogs qui valent le détour

12.1 Livres et autres matériels

12.1.1 Rétro-ingénierie

- Eldad Eilam, *Reversing: Secrets of Reverse Engineering*, (2005)
- Bruce Dang, Alexandre Gazet, Elias Bachaalany, Sebastien Josse, *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, (2014)
- Michael Sikorski, Andrew Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, (2012)
- Chris Eagle, *IDA Pro Book*, (2011)
- Reginald Wong, *Mastering Reverse Engineering: Re-engineer your ethical hacking skills*, (2018)

(Obsolète, mais toujours intéressant) Pavol Cerven, *Crackproof Your Software: Protect Your Software Against Crackers*, (2002).

Également, les livres de Kris Kaspersky.

12.1.2 Windows

- Mark Russinovich, *Microsoft Windows Internals*
- Peter Ferrie – The "Ultimate" Anti-Debugging Reference[1]

Blogs:

- Microsoft: Raymond Chen

[1] http://pferrie.host22.com/papers/antidebug.pdf

- nynaeve.net

12.1.3 C/C++

- Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)
- *ISO/IEC 9899:TC3 (C C99 standard)*, (2007)[2]
- Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)
- C++11 standard[3]
- Agner Fog, *Optimizing software in C++* (2015)[4]
- Marshall Cline, *C++ FAQ*[5]
- Dennis Yurichev, *C/C++ programming language notes*[6]
- JPL Institutional Coding Standard for the C Programming Language[7]

12.1.4 Architecture x86 / x86-64

- Manuels Intel[8]
- Manuels AMD[9]
- Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)[10]
- Agner Fog, *Calling conventions* (2015)[11]
- *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, (2014)
- *Software Optimization Guide for AMD Family 16h Processors*, (2013)

Quelque peu vieux, mais toujours intéressant à lire :

Michael Abrash, *Graphics Programming Black Book*, 1997[12] (il est connu pour son travail sur l'optimisation bas niveau pour des projets tels que Windows NT 3.1 et id Quake).

[2] Aussi disponible en http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf
[3] Aussi disponible en http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.
[4] Aussi disponible en http://agner.org/optimize/optimizing_cpp.pdf.
[5] Aussi disponible en http://www.parashift.com/c++-faq-lite/index.html
[6] Aussi disponible en http://yurichev.com/C-book.html
[7] Aussi disponible en https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf
[8] Aussi disponible en http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
[9] Aussi disponible en http://developer.amd.com/resources/developer-guides-manuals/
[10] Aussi disponible en http://agner.org/optimize/microarchitecture.pdf
[11] Aussi disponible en http://www.agner.org/optimize/calling_conventions.pdf
[12] Aussi disponible en https://github.com/jagregory/abrash-black-book

12.1.5 ARM

- Manuels ARM[13]
- *ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)
- [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)][14]
- Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)[15]

12.1.6 Langage d'assemblage

Richard Blum — Professional Assembly Language.

12.1.7 Java

[Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] [16].

12.1.8 UNIX

Eric S. Raymond, *The Art of UNIX Programming*, (2003)

12.1.9 Programmation en général

- Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)
- Henry S. Warren, *Hacker's Delight*, (2002)Certaines personnes disent que les trucs et astuces de ce livre ne sont plus pertinents aujourd'hui, car ils n'étaient valables que pour les CPUs RISC, où les instructions de branchement sont coûteuses. Néanmoins, ils peuvent énormément aider à comprendre l'algèbre booléenne et toutes les mathématiques associées.

12.1.10 Cryptographie

- Bruce Schneier, *Applied Cryptography*, (John Wiley & Sons, 1994)
- (Free) lvh, *Crypto 101*[17]
- (Free) Dan Boneh, Victor Shoup, *A Graduate Course in Applied Cryptography*[18].

[13]Aussi disponible en http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html
[14]Aussi disponible en http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf
[15]Aussi disponible en https://yurichev.com/ref/ARM%20Cookbook%20(1994)/
[16]Aussi disponible en https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf; http://docs.oracle.com/javase/specs/jvms/se7/html/
[17]Aussi disponible en https://www.crypto101.io/
[18]Aussi disponible en https://crypto.stanford.edu/~dabo/cryptobook/

Chapitre 13

Communautés

Il existe deux excellents subreddits liés à la RE[1] (rétro-ingénierie) sur reddit.com : reddit.com/r/ReverseEngineering/ et reddit.com/r/remath (en lien avec la liaison de la RE et des mathématiques).

Il y a également une section sur l'RE sur le site Stack Exchange: reverseengineering.stackexchange.com.

Sur IRC, il y a les canaux ##re et ##asm sur Libera.

[1] Reverse Engineering

Épilogue

13.1 Des questions ?

Pour toute question, n'hésitez pas à m'envoyer un mail :
my emails. Vous avez une suggestion ou une proposition de contenu supplémentaire pour le livre ? N'hésitez pas à envoyer toute correction (grammaire incluse), etc...

Je travaille beaucoup sur cette œuvre, c'est pourquoi les numéros de pages et les numéros de parties peuvent changer rapidement. S'il vous plaît, ne vous fiez pas à ces derniers si vous m'envoyez un email. Il existe une méthode plus simple : faites une capture d'écran de la page, puis dans un éditeur graphique, soulignez l'endroit où il y a une erreur et envoyez-moi l'image. Je la corrigerai plus rapidement. Et si vous êtes familier avec git et LaTeX vous pouvez corriger l'erreur directement dans le code source :

https://beginners.re/src/.

N'hésitez surtout pas à m'envoyer la moindre erreur que vous pourriez trouver, aussi petite soit-elle, même si vous n'êtes pas certain que ce soit une erreur. J'écris pour les débutants après tout, il est donc crucial pour mon travail d'avoir les retours de débutants.

Appendice

.1 x86

.1.1 Terminologie

Commun en 16-bit (8086/80286), 32-bit (80386, etc.), 64-bit.

octet 8-bit. La directive d'assembleur DB est utilisée pour définir les variables et les tableaux d'octets. Les octets sont passés dans les parties 8-bit des registres: AL/BL/CL/DL/AH/BH/CH/DH/SIL/DIL/R*L.

mot 16-bit. directive assembleur DW —”—. Les mots sont passés dans la partie 16-bit des registres:
AX/BX/CX/DX/SI/DI/R*W.

double mot («dword ») 32-bit. directive assembleur DD —”—. Les double mots sont passés dans les registres (x86) ou dans la partie 32-bit de registres (x64). Dans du code 16-bit, les double mots sont passés dans une paire de registres 16-bit.

quadruple mot («qword ») 64-bit. directive assembleur DQ —”—. En environnement 32-bit, les quadruple mots sont passés dans une paire de registres 32-bit.

tbyte (10 bytes) 80-bit ou 10 octets (utilisé pour les registres FPU IEEE 754).

paragraph (16 bytes)—le terme était répandu dans l'environnement MS-DOS.

Des types de données de même taille (BYTE, WORD, DWORD) existent aussi dans l'API Windows.

.1.2 Registres à usage général

Il est possible d'accéder à de nombreux registres par octet ou par mot de 16-bit.

Les vieux CPUs 8-bit (8080) avaient des registres de 16-bit divisés en deux.

Les programmes écrits pour 8080 pouvaient accéder à l'octet bas des registres de 16-bit, à l'octet haut ou au registre 16-bit en entier.

Peut-être que cette caractéristique a été conservée dans le 8086 pour faciliter le portage.

Cette caractéristiques n'est en général pas présente sur les CPUs RISC.

Les registres préfixés par R- sont apparus en x86-64, et ceux préfixés par E-—dans le 80386.

Ainsi, les R-registres sont 64-bit, et les E-registres—32-bit.

8 GPR ont été ajoutés en x86-64: R8-R15.

N.B.: Dans les manuels Intel, les parties octet de ces registres sont préfixées par , e.g.: , mais IDA nomme ces registres en ajoutant le suffixe , e.g.: .

RAX/EAX/AX/AL

Octet d’indice							
7	6	5	4	3	2	1	0
RAX^{x64}							
				EAX			
						AX	
						AH	AL

AKA accumulateur Le résultat d’une fonction est en général renvoyé via ce registre.

RBX/EBX/BX/BL

Octet d’indice							
7	6	5	4	3	2	1	0
RBX^{x64}							
				EBX			
						BX	
						BH	BL

RCX/ECX/CX/CL

Octet d’indice							
7	6	5	4	3	2	1	0
RCX^{x64}							
				ECX			
						CX	
						CH	CL

AKA compteur: il est utilisé dans ce rôle avec les instructions préfixées par REP et aussi dans les instructions de décalage (SHL/SHR/RxL/RxR).

RDX/EDX/DX/DL

Octet d’indice							
7	6	5	4	3	2	1	0
RDX^{x64}							
				EDX			
						DX	
						DH	DL

RSI/ESI/SI/SIL

Octet d’indice							
7	6	5	4	3	2	1	0
RSI^{x64}							
				ESI			
						SI	
							SIL^{x64}

AKA «source index». Utilisé comme source dans les instructions REP MOVSx, REP CMPSx.

RDI/EDI/DI/DIL

<table>
<tr><td colspan="8">Octet d'indice</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">RDI x64</td></tr>
<tr><td colspan="4"></td><td colspan="4">EDI</td></tr>
<tr><td colspan="6"></td><td colspan="2">DI</td></tr>
<tr><td colspan="7"></td><td>DIL x64</td></tr>
</table>

AKA «destination index». Utilisé comme un pointeur sur la destination dans les instructions REP MOVSx, REP STOSx.

R8/R8D/R8W/R8L

<table>
<tr><td colspan="8">Octet d'indice</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">R8</td></tr>
<tr><td colspan="4"></td><td colspan="4">R8D</td></tr>
<tr><td colspan="6"></td><td colspan="2">R8W</td></tr>
<tr><td colspan="7"></td><td>R8L</td></tr>
</table>

R9/R9D/R9W/R9L

<table>
<tr><td colspan="8">Octet d'indice</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">R9</td></tr>
<tr><td colspan="4"></td><td colspan="4">R9D</td></tr>
<tr><td colspan="6"></td><td colspan="2">R9W</td></tr>
<tr><td colspan="7"></td><td>R9L</td></tr>
</table>

R10/R10D/R10W/R10L

<table>
<tr><td colspan="8">Octet d'indice</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">R10</td></tr>
<tr><td colspan="4"></td><td colspan="4">R10D</td></tr>
<tr><td colspan="6"></td><td colspan="2">R10W</td></tr>
<tr><td colspan="7"></td><td>R10L</td></tr>
</table>

R11/R11D/R11W/R11L

Octet d’indice							
7	6	5	4	3	2	1	0
R11							
				R11D			
						R11W	
							R11L

R12/R12D/R12W/R12L

Octet d’indice							
7	6	5	4	3	2	1	0
R12							
				R12D			
						R12W	
							R12L

R13/R13D/R13W/R13L

Octet d’indice							
7	6	5	4	3	2	1	0
R13							
				R13D			
						R13W	
							R13L

R14/R14D/R14W/R14L

Octet d’indice							
7	6	5	4	3	2	1	0
R14							
				R14D			
						R14W	
							R14L

R15/R15D/R15W/R15L

Octet d’indice							
7	6	5	4	3	2	1	0
R15							
				R15D			
						R15W	
							R15L

RSP/ESP/SP/SPL

Octet d'indice							
7	6	5	4	3	2	1	0
RSP							
				ESP			
						SP	
							SPL

AKA pointeur de pile. Pointe en général sur la pile courante excepté dans le cas où il n'est pas encore initialisé.

RBP/EBP/BP/BPL

Octet d'indice							
7	6	5	4	3	2	1	0
RBP							
				EBP			
						BP	
							BPL

AKA frame pointer. Utilisé d'habitude pour les variables locales et accéder aux arguments de la fonction. En lire plus ici: (1.12.1 on page 94).

RIP/EIP/IP

Octet d'indice							
7	6	5	4	3	2	1	0
RIP^{x64}							
				EIP			
						IP	

AKA «instruction pointer» [2]. En général, il pointe toujours sur l'instruction en cours d'exécution. Il ne peut pas être modifié, toutefois, il est possible de faire ceci (ce qui est équivalent) :

```
MOV EAX, ...
JMP EAX
```

Ou:

```
PUSH value
RET
```

CS/DS/ES/SS/FS/GS

Les registres 16-bit contiennent le sélecteur de code (CS), le sélecteur de données (DS), le sélecteur de pile (SS).

[2] Parfois appelé «program counter»

FS dans win32 pointe sur TLS, GS prend ce rôle dans Linux. C'est fait pour accéder plus au TLS et autres structures comme le TIB.
Dans le passé, ces registres étaient utilisés comme registres de segments (11.7 on page 1297).

Registre de flags

AKA EFLAGS.

Bit (masque)	Abréviation (signification)	Description
0 (1)	CF (Carry)	Les instructions CLC/STC/CMC sont utilisées pour mettre/effacer/changer ce flag
2 (4)	PF (Parity)	(1.25.7 on page 303).
4 (0x10)	AF (Adjust)	Existe seulement pour travailler avec les nombres BCD
6 (0x40)	ZF (Zero)	Mettre à 0 si le résultat de la dernière opération est égal à 0.
7 (0x80)	SF (Sign)	
8 (0x100)	TF (Trap)	Utilisé pour le débogage. S'il est mis, une exception est générée après l'exécution de chaque instruction.
9 (0x200)	IF (Interrupt enable)	Est-ce que les interruptions sont activées Les instructions CLI/STI sont utilisées pour activer/désactiver le flag
10 (0x400)	DF (Direction)	Une direction est défini pour les instructions REP MOVSx/CMPSx/LODSx/SCASx Les instructions CLD/STD sont utilisées pour activer/désactiver le flag Voir aussi 3.26 on page 812.
11 (0x800)	OF (Overflow)	
12, 13 (0x3000)	IOPL (I/O privilege level)[i286]	
14 (0x4000)	NT (Nested task)[i286]	
16 (0x10000)	RF (Resume)[i386]	Utilisé pour le débogage. Le CPU ignore le point d'arrêt matériel dans DRx si le flag est mis.
17 (0x20000)	VM (Virtual 8086 mode)[i386]	
18 (0x40000)	AC (Alignment check)[i486]	
19 (0x80000)	VIF (Virtual interrupt)[i586]	
20 (0x100000)	VIP (Virtual interrupt pending)[i586]	
21 (0x200000)	ID (Identification)[i586]	

Tous les autres flags sont réservés.

.1.3 Registres FPU

8 registres de 80-bit fonctionnant comme une pile: ST(0)-ST(7). N.B.: IDA nomme ST(0) simplement en ST. Les nombres sont stockés au format IEEE 754.

Format d'une valeur :

79	78 … 64	63	62 … 0
S	exposant	I	mantisse ou fraction

(S — signe, I — partie entière)

Mot de Contrôle

Registre contrôlant le comportement du FPU.

Bit	Abréviation (signification)	Description
0	IM (Invalid operation Mask)	
1	DM (Denormalized operand Mask)	
2	ZM (Zero divide Mask)	
3	OM (Overflow Mask)	
4	UM (Underflow Mask)	
5	PM (Precision Mask)	
7	IEM (Interrupt Enable Mask)	Exceptions activées, 1 par défaut (désactivées)
8, 9	PC (Precision Control)	00 — 24 bits (REAL4) 10 — 53 bits (REAL8) 11 — 64 bits (REAL10)
10, 11	RC (Rounding Control)	00 — (par défaut) arrondir au plus proche 01 — arrondir vers $-\infty$ 10 — arrondir vers $+\infty$ 11 — arrondir vers 0
12	IC (Infinity Control)	0 — (par défaut) traite $+\infty$ et $-\infty$ comme non signé 1 — respecte à la fois $+\infty$ et $-\infty$

Les flags PM, UM, OM, ZM, DM, IM définissent si une exception est générée en cas d'erreur correspondante.

Mot d'état

Registre en lecture seule.

Bit	Abréviation (signification)	Description
15	B (Busy)	Est-ce que le FPU fait quelque chose (1) ou des résultats sont prêts (0)
14	C3	
13, 12, 11	TOP	pointe sur le registre zéro actuel
10	C2	
9	C1	
8	C0	
7	IR (Interrupt Request)	
6	SF (Stack Fault)	
5	P (Precision)	
4	U (Underflow)	
3	O (Overflow)	
2	Z (Zero)	
1	D (Denormalized)	
0	I (Invalid operation)	

Les bits SF, P, U, O, Z, D, I indiquent les exceptions.

Vous trouverez des précisions à propos de C3, C2, C1, C0 ici: (1.25.7 on page 303).

N.B.: Lorsque ST(x) est utilisé, le FPU ajoute x à TOP (modulo 8) et c'est ainsi qu'il obtient le numéro du registre interne.

Mot Tag

Le registre possède l'information actuelle à propos de l'utilisation des registres de nombres.

Bit	Abréviation (signification)
15, 14	Tag(7)
13, 12	Tag(6)
11, 10	Tag(5)
9, 8	Tag(4)
7, 6	Tag(3)
5, 4	Tag(2)
3, 2	Tag(1)
1, 0	Tag(0)

Chaque tag contient l'information à propos d'un registre FPU physique, pas logique (ST(x)).

Pour chaque tag:

- 00 — Le registre contient une valeur non-zéro
- 01 — Le registre contient 0
- 10 — Le registre contient une valeur particulière (NAN[3], ∞, ou anormale)
- 11 — Le registre est vide

.1.4 Registres SIMD

Registres MMX

8 registres 64-bit: MM0..MM7.

Registres SSE et AVX

SSE: 8 registres 128-bit: XMM0..XMM7. En x86-64 8 autres registres ont été ajoutés: XMM8..XMM15.

AVX est l'extension de tous ces registres à 256 bits.

.1.5 Registres de débogage

Ils sont utilisés pour le contrôle des points d'arrêt matériel (*hardware breakpoints*).

[3]Not a Number

- DR0 — adresse du point d'arrêt #1
- DR1 — adresse du point d'arrêt #2
- DR2 — adresse du point d'arrêt #3
- DR3 — adresse du point d'arrêt #4
- DR6 — la cause de l'arrêt est indiquée ici
- DR7 — les types de point d'arrêt sont mis ici

DR6

Bit (masque)	Description
0 (1)	B0 — le point d'arrêt #1 a été déclenché
1 (2)	B1 — le point d'arrêt #2 a été déclenché
2 (4)	B2 — le point d'arrêt #3 a été déclenché
3 (8)	B3 — le point d'arrêt #4 a été déclenché
13 (0x2000)	BD — tentative de modification d'un des registres DRx. peut être déclenché si GD est activé
14 (0x4000)	BS — point d'arrêt simple (le flag TF a été mis dans EFLAGS). La plus haute priorité. D'autres bits peuvent être mis aussi.
15 (0x8000)	BT (task switch flag)

N.B. Un point d'arrêt simple est un point d'arrêt qui se produit après chaque instruction. Il peut être enclenché en mettant le flag TF dans EFLAGS (.1.2 on page 1326).

DR7

Les types de point d'arrêt sont mis ici.

Bit (masque)	Description
0 (1)	L0 — activer le point d'arrêt #1 pour la tâche courante
1 (2)	G0 — activer le point d'arrêt #1 pour toutes les tâches
2 (4)	L1 — activer le point d'arrêt #2 pour la tâche courante
3 (8)	G1 — activer le point d'arrêt #2 pour toutes les tâches
4 (0x10)	L2 — activer le point d'arrêt #3 pour la tâche courante
5 (0x20)	G2 — activer le point d'arrêt #3 pour toutes les tâches
6 (0x40)	L3 — activer le point d'arrêt #4 pour la tâche courante
7 (0x80)	G3 — activer le point d'arrêt #4 pour toutes les tâches
8 (0x100)	LE — non supporté depuis P6
9 (0x200)	GE — non supporté depuis P6
13 (0x2000)	GD — exception déclenchée si une instruction MOV essaye de modifier un des registres DRx
16,17 (0x30000)	point d'arrêt #1: R/W — type
18,19 (0xC0000)	point d'arrêt #1: LEN — longueur
20,21 (0x300000)	point d'arrêt #2: R/W — type
22,23 (0xC00000)	point d'arrêt #2: LEN — longueur
24,25 (0x3000000)	point d'arrêt #3: R/W — type
26,27 (0xC000000)	point d'arrêt #3: LEN — longueur
28,29 (0x30000000)	point d'arrêt #4: R/W — type
30,31 (0xC0000000)	point d'arrêt #4: LEN — longueur

Le type de point d'arrêt doit être mis comme suit (R/W) :

- 00 — exécution de l'instruction
- 01 — écriture de données
- 10 — lecture ou écriture I/O (non disponible en mode user)
- 11 — à la lecture ou l'écriture de données

N.B.: le type de point d'arrêt est absent pour la lecture de données, en effet.

La longueur du point d'arrêt est mise comme suit (LEN) :

- 00 — un octet
- 01 — deux octets
- 10 — non défini pour le mode 32-bit, huit octets en mode 64-bit
- 11 — quatre octets

.1.6 Instructions

Les instructions marquées avec un (M) ne sont généralement pas générées par le compilateur: si vous rencontrez l'une d'entre elles, il s'agit probablement de code assembleur écrit à la main, ou de fonctions intrinsèques (11.4 on page 1290).

Seules les instructions les plus fréquemment utilisées sont listées ici. Vous pouvez lire 12.1.4 on page 1315 pour une documentation complète.

Devez-vous connaître tous les opcodes des instructions par cœur? Non, seulement ceux qui sont utilisés pour patcher du code (11.2.1 on page 1288). Tout le reste des opcodes n'a pas besoin d'être mémorisé.

Préfixes

LOCK force le CPU à faire un accès exclusif à la RAM dans un environnement multi-processeurs. Par simplification, on peut dire que lorsqu'une instruction avec ce préfixe est exécutée, tous les autres CPU dans un système multi-processeur sont stoppés. Le plus souvent, c'est utilisé pour les sections critiques, les sémaphores et les mutex. Couramment utilisé avec ADD, AND, BTR, BTS, CMPXCHG, OR, XADD, XOR. Vous pouvez en lire plus sur les sections critiques ici (6.5.4 on page 1026).

REP est utilisé avec les instructions MOVSx et STOSx: exécute l'instruction dans une boucle, le compteur est situé dans le registre CX/ECX/RCX. Pour une description plus détaillée de ces instructions, voir MOVSx (.1.6 on page 1334) et STOSx (.1.6 on page 1336).

Les instructions préfixées par REP sont sensibles au flag DF, qui est utilisé pour définir la direction.

REPE/REPNE (AKA REPZ/REPNZ) utilisé avec les instructions CMPSx et SCASx: exécute la dernière instruction dans une boucle, le compteur est mis dans le registre CX/ECX/RCX. Elle s'arrête prématurément si ZF vaut 0 (REPE) ou si ZF vaut 1 (REPNE).

Pour une description plus détaillée de ces instructions, voir CMPSx (.1.6 on page 1338) et SCASx (.1.6 on page 1335).

Les instructions préfixées par REPE/REPNE sont sensibles au flag DF, qui est utilisé pour définir la direction.

Instructions les plus fréquemment utilisées

Celles-ci peuvent être mémorisées en premier.

ADC (*add with carry*) ajoute des valeurs, incrémente le résultat si le flag CF est mis. ADC est souvent utilisé pour ajouter des grandes valeurs, par exemple, pour ajouter deux valeurs 64-bit dans un environnement 32-bit en utilisant deux instructions, ADD et ADC. Par exemple:

```
; fonctionne avec des valeurs 64-bit: ajoute val1 à val2.
; .lo signifie les plus bas des 32 bits, .hi signifie les plus hauts.
ADD val1.lo, val2.lo
ADC val1.hi, val2.hi ; utilise CF mis à 0 ou 1 par l'instruction
    précédente
```

Un autre exemple: 1.34 on page 509.

ADD ajoute deux valeurs

AND «et» logique

CALL appelle une autre fonction:
`PUSH address_after_CALL_instruction; JMP label`

CMP compare les valeurs et met les flags, comme SUB mais sans écrire le résultat

DEC décrémente. Contrairement aux autres instructions arithmétiques, DEC ne modifie pas le flag CF.

IMUL multiplication signée IMUL est souvent utilisé à la place de MUL, voir ici: 11.1 on page 1287.

INC incrémente. Contrairement aux autres instructions arithmétiques, INC ne modifie pas le flag CF.

JCXZ, JECXZ, JRCXZ (M) saute si CX/ECX/RCX=0

JMP saute à une autre adresse. L'opcode a un jump offset.

Jcc (où cc — condition code)

Beaucoup de ces instructions ont des synonymes (notés avec AKA), qui ont été ajoutés par commodité. Ils sont codés avec le même opcode. L'opcode a un offset de saut.

JAE AKA JNC: saut si supérieur ou égal (non signé) : C=0

JA AKA JNBE: saut si supérieur (non signé) : CF=0 et ZF=0

JBE saut si inférieur ou égal (non signé) : CF=1 ou ZF=1

JB AKA JC: saut si inférieur(non signé) : CF=1

JC AKA JB: saut si CF=1

JE AKA JZ: saut si égal ou zéro: ZF=1

JGE saut si supérieur ou égal (signé) : SF=OF

JG saut si supérieur (signé) : ZF=0 et SF=OF

JLE saut si inférieur ou égal (signé) : ZF=1 ou SF≠OF

JL saut si inférieur (signé) : SF≠OF

JNAE AKA JC: saut si non supérieur ou égal (non signé) : CF=1

JNA saut si non supérieur (non signé) : CF=1 et ZF=1

JNBE saut si non inférieur ou égal (non signé) : CF=0 et ZF=0

JNB AKA JNC: saut si non inférieur (non signé) : CF=0

JNC AKA JAE: saut si CF=0, synonyme de JNB.

JNE AKA JNZ: saut si non égal ou non zéro: ZF=0

JNGE saut si non supérieur ou égal (signé) : SF≠OF

JNG saut si non supérieur (signé) : ZF=1 ou SF≠OF

JNLE saut si non inférieur ou égal (signé) : ZF=0 et SF=OF

JNL saut si non inférieur (signé) : SF=OF

JNO saut si non débordement: OF=0

JNS saut si le flag SF vaut zéro

JNZ AKA JNE: saut si non égal ou non zéro: ZF=0

JO saut si débordement: OF=1

JPO saut si le flag PF vaut 0 (Jump Parity Odd)

JP AKA JPE : saut si le flag PF est mis

JS saut si le flag SF est mis

JZ AKA JE: saut si égal ou zéro: ZF=1

LAHF copie certains bits du flag dans AH:

7	6		4		2		0
SF	ZF		AF		PF		CF

Cette instruction est souvent utilisée dans du code relatif au FPU.

LEAVE équivalente à la paire d'instructions `MOV ESP, EBP` et `POP EBP` — autrement dit, cette instruction remet le pointeur de pile et restaure le registre EBP à l'état initial.

LEA (*Load Effective Address*) forme une adresse

Cette instruction n'a pas été conçue pour sommer des valeurs et/ou les multiplier, mais pour former une adresse, e.g., pour calculer l'adresse d'un élément d'un tableau en ajoutant l'adresse du tableau, l'index de l'élément multiplié par la taille de l'élément[4].

Donc, la différence entre MOV et LEA est que MOV forme une adresse mémoire et charge une valeur depuis la mémoire ou l'y stocke, alors que LEA forme simplement une adresse.

Mais néanmoins, elle peut être utilisée pour tout autre calcul.

LEA est pratique car le calcul qu'elle effectue n'altère pas les flags du CPU. Ceci peut être très important pour les processeurs OOE (afin de créer moins de dépendances). À part ça, au moins à partir du Pentium, l'instruction LEA est exécutée en 1 cycle.

```
int f(int a, int b)
{
        return a*8+b;
};
```

Listing 1 : MSVC 2010 avec optimisation

```
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_f      PROC
        mov     eax, DWORD PTR _b$[esp-4]
        mov     ecx, DWORD PTR _a$[esp-4]
        lea     eax, DWORD PTR [eax+ecx*8]
        ret     0
_f      ENDP
```

Intel C++ utilise encore plus LEA:

```
int f1(int a)
{
        return a*13;
};
```

Listing 2 : Intel C++ 2011

```
_f1     PROC NEAR
        mov       ecx, DWORD PTR [4+esp]      ; ecx = a
        lea       edx, DWORD PTR [ecx+ecx*8]  ; edx = a*9
        lea       eax, DWORD PTR [edx+ecx*4]  ; eax = a*9 + a*4 = a*13
        ret
```

Ces deux instructions sont plus rapide qu'un IMUL.

[4]Voir aussi: Wikipédia

MOVSB/MOVSW/MOVSD/MOVSQ copier l'octet/ le mot 16-bit/ le mot 32-bit/ le mot 64-bit depuis l'adresse se trouvant dans SI/ESI/RSI vers celle se trouvant dans DI/EDI/RDI.

Avec le préfixe REP, elle est répétée en boucle, le compteur étant stocker dans le registre CX/ECX/RCX: ça fonctionne comme memcpy() en C. Si la taille du bloc est connue pendant la compilation, memcpy() est souvent mise en ligne dans un petit morceau de code en utilisant REP MOVSx, parfois même avec plusieurs instructions.

L'équivalent de memcpy(EDI, ESI, 15) est:

```
; copier 15 octets de ESI vers EDI
CLD         ; mettre la direction à en avant
MOV ECX, 3
REP MOVSD   ; copier 12 octets
MOVSW       ; copier 2 octets de plus
MOVSB       ; copier l'octet restant
```

(Apparemment, c'est plus rapide que de copier 15 octets avec un seul REP MOVSB).

MOVSX charger avec extension du signe voir aussi: (1.23.1 on page 264)

MOVZX icharger et effacer tous les autres bits voir aussi: (1.23.1 on page 265)

MOV charger une valeur. Le nom de cette instruction est inapproprié, ce qui entraîne des confusions (la donnée n'est pas déplacée, mais copiée), dans d'autres architectures la même instruction est en général appelée « LOAD » et/ou « STORE » ou quelque chose comme ça.

Une chose importante: si vous mettez la partie 16-bit basse d'un registre 32-bit en mode 32-bit, les 16-bit haut restent comme ils étaient. Mais si vous modifiez la partie 32-bit basse d'un registre en mode 64-bit, les 32-bits haut du registre seront mis à zéro.

Peut-être que ça a été fait pour simplifier le portage du code sur x86-64.

MUL multiplier sans signe. `IMUL` est souvent utilisée au lieu de `MUL`, en lire plus ici: 11.1 on page 1287.

NEG négation: $op = -op$ La même chose que `NOT op / ADD op, 1`.

NOP NOP. Son opcode est 0x90, qui est en fait l'instruction sans effet `XCHG EAX,EAX`. Ceci implique que le x86 n'a pas d'instruction NOP dédiée (comme dans de nombreux RISC). Ce livre contient au moins un listing où GDB affiche NOP comme l'instruction 16-bit XCHG: 1.11.1 on page 68.

Plus d'exemples de telles opérations: (.1.7 on page 1347).

NOP peut être généré par le compilateur pour aligner des labels sur une limite de 16-octets. Un autre usage très répandu de NOP est de remplacer manuellement (patcher) une instruction, comme un saut conditionnel, par NOP, afin de désactiver cette exécution.

NOT op1: $op1 = \neg op1$. inversion logique Caractéristique importante—l'instruction ne change pas les flags.

OR «ou» logique

POP prend une valeur depuis la pile: `value=SS:[ESP]; ESP=ESP+4 (ou 8)`

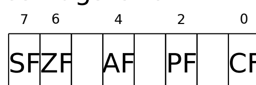
PUSH pousse une valeur sur la pile: `ESP=ESP-4 (ou 8); SS:[ESP]=value`

RET Revient d'une sous-routine: `POP tmp; JMP tmp`.

En fait, RET est une macro du langage d'assemblage, sous les environnements Windows et *NIX, elle est traduite en RETN («return near») ou, du temps de MS-DOS, où la mémoire était adressée différemment (11.7 on page 1297), en RETF («return far»).

RET peut avoir un opérande. Alors il fonctionne comme ceci:
`POP tmp; ADD ESP op1; JMP tmp`. RET avec un opérande termine en général les fonctions avec la convention d'appel *stdcall*, voir aussi: 6.1.2 on page 953.

SAHF copier des bits de AH vers les flags CPU:



Cette instruction est souvent utilisée dans du code relatif au FPU.

SBB (*subtraction with borrow*) soustrait les valeurs, décrémente le résultat si le flag CF est mis. SBB est souvent utilisé pour la soustraction de grandes valeurs, par exemple:

```
; fonctionne avec des valeurs 64-bit: soustrait val2 de val1.
; .lo signifie les 32 bits les plus bas, .hi signifie les plus hauts.
SUB val1.lo, val2.lo
SBB val1.hi, val2.hi ; utilise CF mis à 1 ou 0 par l'instruction
    précédente
```

Un autre exemple: 1.34 on page 509.

SCASB/SCASW/SCASD/SCASQ (M) compare un octet/ un mot 16-bit/ un mot 32-bit/ un mot 64-bit stocké dans AX/EAX/RAX avec une variable dont l'adresse est dans DI/EDI/RDI. Met les flags comme le fait CMP.

Cette instruction est souvent utilisée avec le préfixe REPNE: continue de scanner le buffer jusqu'à ce qu'une valeur particulière stockée dans AX/EAX/RAX soit trouvée. D'où le «NE» dans REPNE: continue de scanner tant que les valeurs comparées ne sont pas égales et s'arrête lorsqu'elles le sont.

Elle est souvent utilisée comme la fonction C standard strlen(), pour déterminer la longueur d'une chaîne ASCIIZ :

Exemple:

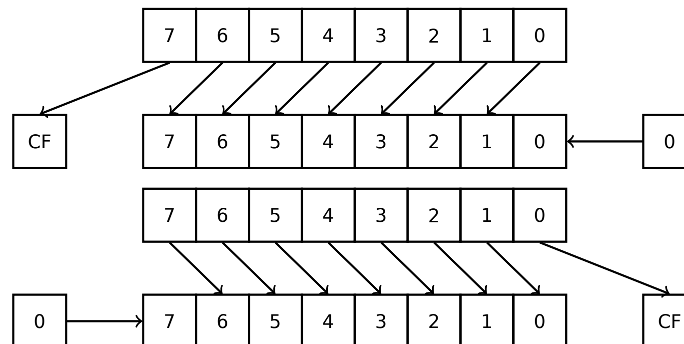
```
lea     edi, string
mov     ecx, 0FFFFFFFFh ; scanne les octets 2^32 - 1, i.e., presque
    l'infini
xor     eax, eax        ; 0 est le terminateur
repne scasb
```

```
add     edi, 0FFFFFFFFh ; le corriger

; maintenant EDI pointe sur le dernier caractère de la chaîne ASCIIZ.

; déterminer la longueur de la chaîne
; ECX vaut = -1-strlen

not     ecx
dec     ecx

; maintenant ECX contient la longueur de la chaîne
```

Si nous utilisons une valeur différente dans AX/EAX/RAX, la fonction se comporte comme la fonction C standard memchr(), i.e., elle trouve un octet spécifique.

SHL décale une valeur à gauche

SHR décale une valeur à droite:



Ces instructions sont utilisées fréquemment pour la multiplication et la division par 2^n. Une autre utilisation très fréquente est le traitement des champs de bits: 1.28 on page 392.

SHRD op1, op2, op3: décale la valeur dans op2 de op3 bits vers la droite, en prenant les bits depuis op1.

Exemple: 1.34 on page 509.

STOSB/STOSW/STOSD/STOSQ stocke un octet/ un mot 16-bit/ un mot 32-bit/ un mot 64-bit de AX/EAX/RAX à l'adresse se trouvant dans DI/EDI/RDI.

Couplée avec le préfixe REP, elle est répétée en boucle, le compteur étant dans le registre CX/ECX/RCX: elle fonctionne comme memset() en C. Si la taille du bloc est connue lors de la compilation, memset() est souvent mise en ligne dans un petit morceau de code en utilisant REP MOVSx, parfois même avec plusieurs instructions.

memset(EDI, 0xAA, 15) est équivalent à:

```
; sotcke 15 octets 0xAA dans EDI
CLD                 ; met la direction à en avant
MOV EAX, 0AAAAAAAAh
```

```
MOV ECX, 3
REP STOSD           ; écrit 12 octets
STOSW               ; écrit 2 octets de plus
STOSB               ; écrit l'octet restant
```

(Apparemment, ça fonctionne plus vite que de de stocker 15 octets avec un seul REP STOSB).

SUB soustrait des valeurs. Une utilisation fréquente est `SUB reg,reg`, qui met *reg* à zéro.

TEST comme AND mais sans sauvegarder le résultat, voir aussi: 1.28 on page 392

XOR op1, op2: XOR[5] valeurs. $op1 = op1 \oplus op2$. Un schéma récurrent est `XOR reg,reg`, qui met *reg* à zéro.

Instructions les moins fréquemment utilisées

BSF *bit scan forward*, voir aussi: 1.36.2 on page 543

BSR *bit scan reverse*

BSWAP *(byte swap)*, change le boutisme de la valeur.

BTC bit test and complement

BTR bit test and reset

BTS bit test and set

BT bit test

CBW/CWD/CWDE/CDQ/CDQE Étendre le signe de la valeur:

CBW Convertit l'octet dans AL en un mot dans AX

CWD Convertit le mot dans AX en double-mot dans DX:AX

CWDE Convertit le mot dans AX en double-mot dans EAX

CDQ Convertit le double-mot dans EAX en quadruple-mot dans EDX:EAX

CDQE (x64) Convertit le double-mot dans EAX en quadruple-mot dans RAX

Cette instruction examine le signe de la valeur, l'étend à la partie haute de la valeur nouvellement construite. Voir aussi: 1.34.5 on page 521.

Il est intéressant de savoir que ces instructions furent initialement appelées SEX (*Sign EXtend*), comme l'écrit Stephen P. Morse (un des concepteurs du CPU 8086) dans [Stephen P. Morse, *The 8086 Primer*, (1980)][6] :

> The process of stretching numbers by extending the sign bit is called sign extension. The 8086 provides instructions (Fig. 3.29) to

[5] eXclusive OR (OU exclusif)
[6] Aussi disponible en `https://archive.org/details/The8086Primer`

facilitate the task of sign extension. These instructions were initially named SEX (sign extend) but were later renamed to the more conservative CBW (convert byte to word) and CWD (convert word to double word).

CLD efface le flag DF.

CLI (M) efface le flag IF.

CLC (M) efface le flag CF

CMC (M) bascule le flag CF

CMOVcc MOV conditionnel: charge si la condition est vraie. Les codes condition sont les même que l'instruction Jcc (.1.6 on page 1331).

CMPSB/CMPSW/CMPSD/CMPSQ (M) compare un octet/ mot de 16-bit/ mot de 32-bit/ mot de 64-bit à partir de l'adresse qui se trouve dans SI/ESI/RSI avec la variable à l'adresse stockée dans DI/EDI/RDI.

Avec le préfixe REP, elle est répétée en boucle, le compteur est stocké dans le registre CX/ECX/RCX, le processus se répétera jusqu'à ce que le flag ZF soit zéro (i.e., jusqu'à ce que les valeurs soient égales l'une à l'autre, d'où le «E» dans REPE).

Ca fonctionne comme memcmp() en C.

Exemple tiré du noyau de Windows NT (WRK v1.2) :

Listing 3 : base\ntos\rtl\i386\movemem.asm

```
; ULONG
; RtlCompareMemory (
; IN PVOID Source1,
; IN PVOID Source2,
; IN ULONG Length
; )
;
; Routine Description:
;
; This function compares two blocks of memory and returns the number
; of bytes that compared equal.
;
; Arguments:
;
; Source1 (esp+4) - Supplies a pointer to the first block of memory to
; compare.
;
; Source2 (esp+8) - Supplies a pointer to the second block of memory to
; compare.
;
; Length (esp+12) - Supplies the Length, in bytes, of the memory to be
; compared.
;
; Return Value:
```

```
;
; The number of bytes that compared equal is returned as the function
; value. If all bytes compared equal, then the length of the original
; block of memory is returned.
;
;--

RcmSource1      equ     [esp+12]
RcmSource2      equ     [esp+16]
RcmLength       equ     [esp+20]

CODE_ALIGNMENT
cPublicProc _RtlCompareMemory,3
cPublicFpo 3,0

        push    esi                     ; save registers
        push    edi                     ;
        cld                             ; clear direction
        mov     esi,RcmSource1          ; (esi) -> first block to
    compare
        mov     edi,RcmSource2          ; (edi) -> second block to
    compare

;
; Compare dwords, if any.
;

rcm10:  mov     ecx,RcmLength           ; (ecx) = length in bytes
        shr     ecx,2                   ; (ecx) = length in dwords
        jz      rcm20                   ; no dwords, try bytes
        repe    cmpsd                   ; compare dwords
        jnz     rcm40                   ; mismatch, go find byte

;
; Compare residual bytes, if any.
;

rcm20:  mov     ecx,RcmLength           ; (ecx) = length in bytes
        and     ecx,3                   ; (ecx) = length mod 4
        jz      rcm30                   ; 0 odd bytes, go do dwords
        repe    cmpsb                   ; compare odd bytes
        jnz     rcm50                   ; mismatch, go report how far we
    got

;
; All bytes in the block match.
;

rcm30:  mov     eax,RcmLength           ; set number of matching bytes
        pop     edi                     ; restore registers
        pop     esi                     ;
        stdRET  _RtlCompareMemory

;
```

```
; When we come to rcm40, esi (and edi) points to the dword after the
; one which caused the mismatch. Back up 1 dword and find the byte.
; Since we know the dword didn't match, we can assume one byte won't.
;

rcm40:  sub     esi,4                   ; back up
        sub     edi,4                   ; back up
        mov     ecx,5                   ; ensure that ecx doesn't count
    out
        repe    cmpsb                   ; find mismatch byte

;
; When we come to rcm50, esi points to the byte after the one that
; did not match, which is TWO after the last byte that did match.
;

rcm50:  dec     esi                     ; back up
        sub     esi,RcmSource1          ; compute bytes that matched
        mov     eax,esi                 ;
        pop     edi                     ; restore registers
        pop     esi                     ;
        stdRET  _RtlCompareMemory

stdENDP _RtlCompareMemory
```

N.B.: cette fonction utilise une comparaison 32-bit (CMPSD) si la taille du bloc est un multiple de 4, ou sinon une comparaison par octet (CMPSB).

CPUID renvoie des informations sur les fonctionnalités du CPU. Voir aussi: (1.30.6 on page 475).

DIV division non signée

IDIV division signée

INT (M) : `INT x` est similaire à `PUSHF; CALL dword ptr [x*4]` en environnement 16-bit. Elle était énormément utilisée dans MS-DOS, fonctionnant comme un vecteur syscall. Les registres AX/BX/CX/DX/SI/DI étaient remplis avec les arguments et le flux sautait à l'adresse dans la table des vecteurs d'interruption (Interrupt Vector Table, située au début de l'espace d'adressage). Elle était répandue car INT a un opcode court (2 octets) et le programme qui a besoin d'un service MS-DOS ne doit pas déterminer l'adresse du point d'entrée de ce service. Le gestionnaire d'interruption renvoie le contrôle du flux à l'appelant en utilisant l'instruction IRET.

Le numéro d'interruption les plus utilisé était 0x21, servant une grande partie de on API. Voir aussi: [Ralf Brown *Ralf Brown's Interrupt List*], pour les listes d'interruption plus exhaustives et d'autres informations sur MS-DOS.

Durant l'ère post-MS-DOS, cette instruction était toujours utilisée comme un syscall à la fois dans Linux et Windows (6.3 on page 972), mais fût remplacée plus tard par les instructions SYSENTER ou SYSCALL.

INT 3 (M) : cette instruction est proche de `INT`, elle a son propre opcode d'1 octet (`0xCC`), et est très utilisée pour le débogage. Souvent, les débogueurs écrivent

simplement l'octet 0xCC à l'adresse du point d'arrêt à mettre, et lorsqu'une exception est levée, l'octet original est restauré et l'instruction originale à cette adresse est ré-exécutée.
Depuis Windows NT, une exception EXCEPTION_BREAKPOINT est déclenchée lorsque le CPU exécute cette instruction. Cet évènement de débogage peut être intercepté et géré par un débogueur hôte, si il y en a un de chargé. S'il n'y en a pas de charger, Windows propose de lancer un des débogueurs enregistré dans le système. Si MSVS[7] est installé, son débogueur peut être chargé et connecté au processus. Afin de protéger contre le reverse engineering, de nombreuses méthodes anti-débogage vérifient l'intégrité du code chargé.

MSVC possède une fonction intrinsèque pour l'instruction: __debugbreak()[8].

Il y a aussi une fonction win32 dans kernel32.dll appelée DebugBreak()[9], qui exécute aussi INT 3.

IN (M) lire des données depuis le port. On trouve cette instruction dans les drivers de l'OS ou dans de l'ancien code MS-DOS, par exemple (8.8.3 on page 1105).

IRET : était utilisée dans l'environnement MS-DOS pour retourner d'un gestionnaire d'interruption appelé par l'instruction INT. Équivalent à POP tmp; POPF; JMP tmp.

LOOP (M) décrémente CX/ECX/RCX, saute si il est toujours non zéro.

L'instruction LOOP était souvent utilisée dans le code DOS qui travaillait avec des dispositifs externes. Pour ajouter un petit délai, on utilisait ceci:

```
        MOV     CX, nnnn
LABEL:  LOOP    LABEL
```

Le défaut est évident: le délai dépend de la vitese du CPU.

OUT (M) encoie des données sur le port. L'instruction peut être vue, en général, dans les drivers d'OS ou dans du vieux code MS-DOS, par exemple (8.8.3 on page 1105).

POPA (M) restaure les valeurs des registres (R|E)DI, (R|E)SI, (R|E)BP, (R|E)BX, (R|E)DX, (R|E)CX, (R|E)AX depuis la pile.

POPCNT population count. Compte le nombre de bits à 1 dans la valeur.

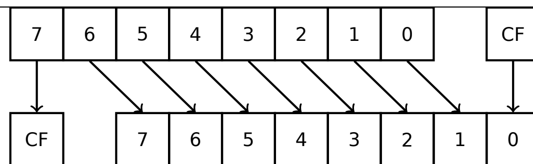
POPF restaure les flags depuis la pile (AKA registre EFLAGS)

PUSHA (M) pousse les valeurs des registres (R|E)AX, (R|E)CX, (R|E)DX, (R|E)BX, (R|E)BP, (R|E)SI, (R|E)DI sur la pile.

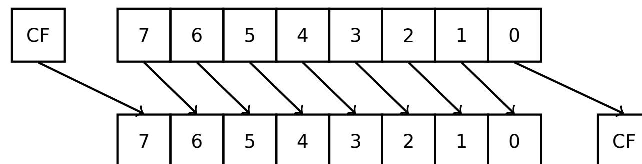
PUSHF pousse les flags (AKA registre EFLAGS)

RCL (M) pivote vers la gauche via le flag CF:

[7] Microsoft Visual Studio
[8] MSDN
[9] MSDN

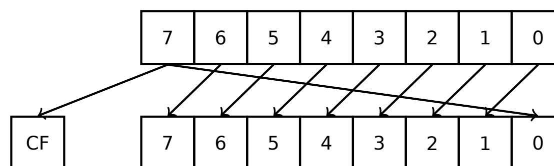


RCR (M) pivote vers la droite via le flag CF:

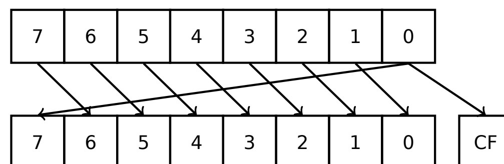


ROL/ROR (M) décalage cyclique

ROL: rotation à gauche:



ROR: rotation à droite:

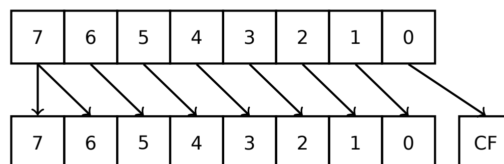


En dépit du fait que presque presque tous les CPUs aient ces instructions, il n'y a pas d'opération correspondante en C/C++, donc les compilateurs de ces LPs ne génèrent en général pas ces instructions.

Par commodité pour le programmeur, au moins MSVC fourni les pseudo-fonctions (fonctions intrinsèques du compilateur) *_rotl()* et *_rotr()*[10], qui sont traduites directement par le compilateur en ces instructions.

SAL Décalage arithmétique à gauche, synonyme de SHL

SAR Décalage arithmétique à droite



De ce fait, le bit de signe reste toujours à la place du MSB.

[10] MSDN

SETcc op: charge 1 dans l'opérande (octet seulement) si la condition est vraie et zéro sinon. Les codes conditions sont les même que les instructions Jcc (.1.6 on page 1331).

STC (M) met le flag CF

STD (M) Met le flag DF. Cette instruction n'est pas générée par les compilateurs et est en général rare. Par exemple, elle peut être trouvée dans le fichier du noyau de Windows `ntoskrnl.exe`, dans la routine de copie mémoire écrite à la main.

STI (M) met le flag IF

SYSCALL (AMD) appelle un appel système (6.3 on page 972)

SYSENTER (Intel) appel un appel système (6.3 on page 972)

UD2 (M) instruction indéfinie, lève une exception. Utilisée pour tester.

XCHG (M) échange les valeurs dans les opérandes

Cette instruction est rare: les compilateurs ne la génère pas, car à partir du Pentium, XCHG avec comme opérande une adresse en mémoire s'exécute comme si elle avait le préfixe LOCK ([Michael Abrash, *Graphics Programming Black Book*, 1997chapter 19]). Peut-être que les ingénieurs d'Intel ont fait cela pour la compatibilité avec les primitives de synchronisation. Ainsi, à partir du Pentium, XCHG peut être lente. D'un autre côté, XCHG était très populaire chez les programmeurs en langage d'assemblage. Donc, si vous voyez XCHG dans le code, ça peut être un signe que ce morceau de code a été écrit à la main. Toutefois, au moins le compilateur Borland Delphi génère cette instruction.

Instructions FPU

Le suffixe -R dans le mnémonique signifie en général que les opérandes sont inversés, le suffixe -P implique qu'un élément est supprimé de la pile après l'exécution de l'instruction, le suffixe -PP implique que deux éléments sont supprimés.

Les instructions -P sont souvent utiles lorsque nous n'avons plus besoin que la valeur soit présente dans la pile FPU après l'opération.

FABS remplace la valeur dans ST(0) par sa valeur absolue

FADD op: ST(0)=op+ST(0)

FADD ST(0), ST(i) : ST(0)=ST(0)+ST(i)

FADDP ST(1)=ST(0)+ST(1); supprime un élément de la pile, i.e., les valeurs sur la pile sont remplacées par leurs somme

FCHS ST(0)=-ST(0)

FCOM compare ST(0) avec ST(1)

FCOM op: compare ST(0) avec op

FCOMP compare ST(0) avec ST(1); supprime un élément de la pile

FCOMPP compare ST(0) avec ST(1); supprime deux éléments de la pile

FDIVR op: ST(0)=op/ST(0)

FDIVR ST(i), ST(j) : ST(i)=ST(j)/ST(i)

FDIVRP op: ST(0)=op/ST(0) ; supprime un élément de la pile

FDIVRP ST(i), ST(j) : ST(i)=ST(j)/ST(i) ; supprime un élément de la pile

FDIV op: ST(0)=ST(0)/op

FDIV ST(i), ST(j) : ST(i)=ST(i)/ST(j)

FDIVP ST(1)=ST(0)/ST(1) ; supprime un élément de la pile, i.e, les valeurs du dividende et du diviseur sont remplacées par le quotient

FILD op: convertit un entier n et le pousse sur la pile.

FIST op: convertit la valeur dans ST(0) en un entier dans op

FISTP op: convertit la valeur dans ST(0) en un entier dans op; supprime un élément de la pile

FLD1 pousse 1 sur la pile

FLDCW op: charge le FPU control word (.1.3 on page 1327) depuis le 16-bit op.

FLDZ pousse zéro sur la pile

FLD op: pousse op sur la pile.

FMUL op: ST(0)=ST(0)*op

FMUL ST(i), ST(j) : ST(i)=ST(i)*ST(j)

FMULP op: ST(0)=ST(0)*op; supprime un élément de la pile

FMULP ST(i), ST(j) : ST(i)=ST(i)*ST(j) ; supprime un élément de la pile

FSINCOS : tmp=ST(0) ; ST(1)=sin(tmp) ; ST(0)=cos(tmp)

FSQRT : $ST(0) = \sqrt{ST(0)}$

FSTCW op: stocker le mot de contrôle FPU (.1.3 on page 1327) dans l'op 16-bit après avoir vérifié s'il y a des exceptions en attente.

FNSTCW op: stocker le mot de contrôle FPU (.1.3 on page 1327) dans l'op 16-bit.

FSTSW op: stocker le mot d'état FPU (.1.3 on page 1327) dans l'op 16-bit après avoir vérifié s'il y a des exceptions en attente.

FNSTSW op: stocker le mot d'état FPU (.1.3 on page 1327) dans l'op 16-bit.

FST op: copie ST(0) dans op

FSTP op: copie ST(0) dans op; supprime un élément de la pile

FSUBR op: ST(0)=op-ST(0)

FSUBR ST(0), ST(i) : ST(0)=ST(i)-ST(0)

FSUBRP ST(1)=ST(0)-ST(1) ; supprime un élément de la pile, i.e., la valeur dans la pile est remplacée par la différence

FSUB op: ST(0)=ST(0)-op

FSUB ST(0), ST(i) : ST(0)=ST(0)-ST(i)

FSUBP ST(1)=ST(1)-ST(0) ; supprime un élément de la pile, i.e., la valeur dans la pile est remplacée par la différence

FUCOM ST(i) : compare ST(0) et ST(i)

FUCOM compare ST(0) et ST(1)

FUCOMP compare ST(0) et ST(1) ; supprime un élément de la pile.

FUCOMPP compare ST(0) et ST(1) ; supprime deux éléments de la pile.

L'instruction se comporte comme FCOM, mais une exception est levée seulement si un opérande est SNaN, tandis que les nombres QNaN sont traités normalement.

FXCH ST(i) échange les valeurs dans ST(0) et ST(i)

FXCH échange les valeurs dans ST(0) et ST(1)

Instructions ayant un opcode affichable en ASCII

(En mode 32-bit.)

Elles peuvent être utilisées pour la création de shellcode. Voir aussi: 8.14.1 on page 1178.

caractère ASCII	code hexadécimal	instruction x86
0	30	XOR
1	31	XOR
2	32	XOR
3	33	XOR
4	34	XOR
5	35	XOR
7	37	AAA
8	38	CMP
9	39	CMP
:	3a	CMP
;	3b	CMP
<	3c	CMP
=	3d	CMP
?	3f	AAS
@	40	INC
A	41	INC
B	42	INC
C	43	INC
D	44	INC
E	45	INC
F	46	INC
G	47	INC
H	48	DEC
I	49	DEC

J	4a	DEC
K	4b	DEC
L	4c	DEC
M	4d	DEC
N	4e	DEC
O	4f	DEC
P	50	PUSH
Q	51	PUSH
R	52	PUSH
S	53	PUSH
T	54	PUSH
U	55	PUSH
V	56	PUSH
W	57	PUSH
X	58	POP
Y	59	POP
Z	5a	POP
[	5b	POP
\	5c	POP
]	5d	POP
^	5e	POP
_	5f	POP
`	60	PUSHA
a	61	POPA
f	66	en mode 32-bit, change pour une taille d'opérande de 16-bit
g	67	en mode 32-bit, change pour une taille d'adresse 16-bit
h	68	PUSH
i	69	IMUL
j	6a	PUSH
k	6b	IMUL
p	70	JO
q	71	JNO
r	72	JB
s	73	JAE
t	74	JE
u	75	JNE
v	76	JBE
w	77	JA
x	78	JS
y	79	JNS
z	7a	JP

De même:

caractère ASCII	code hexadécimal	instruction x86

f	66	en mode 32-bit, change pour une taille d'opérande de 16-bit
g	67	en mode 32-bit, change pour une taille d'adresse 16-bit

En résumé: AAA, AAS, CMP, DEC, IMUL, INC, JA, JAE, JB, JBE, JE, JNE, JNO, JNS, JO, JP, JS, POP, POPA, PUSH, PUSHA, XOR.

.1.7 npad

C'est une macro du langage d'assemblage pour aligner les labels sur une limite spécifique.

C'est souvent nécessaire pour des labels très utilisés, comme par exemple le début d'un corps de boucle. Ainsi, le CPU peut charger les données ou le code depuis la mémoire efficacement, à travers le bus mémoire, les caches, etc.

Pris de `listing.inc` (MSVC) :

À propos, c'est un exemple curieux des différentes variations de `NOP`. Toutes ces instructions n'ont pas d'effet, mais ont une taille différente.

Avoir une seule instruction sans effet au lieu de plusieurs est accepté comme étant meilleur pour la performance du CPU.

```
;; LISTING.INC
;;
;; This file contains assembler macros and is included by the files created
;; with the -FA compiler switch to be assembled by MASM (Microsoft Macro
;; Assembler).
;;
;; Copyright (c) 1993-2003, Microsoft Corporation. All rights reserved.

;; non destructive nops
npad macro size
if size eq 1
  nop
else
 if size eq 2
   mov edi, edi
 else
  if size eq 3
    ; lea ecx, [ecx+00]
    DB 8DH, 49H, 00H
  else
   if size eq 4
     ; lea esp, [esp+00]
     DB 8DH, 64H, 24H, 00H
   else
    if size eq 5
```

```
      add eax, DWORD PTR 0
    else
     if size eq 6
       ; lea ebx, [ebx+00000000]
       DB 8DH, 9BH, 00H, 00H, 00H, 00H
     else
      if size eq 7
        ; lea esp, [esp+00000000]
        DB 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
      else
       if size eq 8
        ; jmp .+8; .npad 6
        DB 0EBH, 06H, 8DH, 9BH, 00H, 00H, 00H, 00H
       else
        if size eq 9
         ; jmp .+9; .npad 7
         DB 0EBH, 07H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
        else
         if size eq 10
          ; jmp .+A; .npad 7; .npad 1
          DB 0EBH, 08H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 90H
         else
          if size eq 11
           ; jmp .+B; .npad 7; .npad 2
           DB 0EBH, 09H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8BH, 0FFH
          else
           if size eq 12
            ; jmp .+C; .npad 7; .npad 3
            DB 0EBH, 0AH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 49H, 00H
           else
            if size eq 13
             ; jmp .+D; .npad 7; .npad 4
             DB 0EBH, 0BH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 64H, 24↙
    ↳ H, 00H
            else
             if size eq 14
              ; jmp .+E; .npad 7; .npad 5
              DB 0EBH, 0CH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 05H, 00H, ↙
    ↳ 00H, 00H, 00H
             else
              if size eq 15
               ; jmp .+F; .npad 7; .npad 6
               DB 0EBH, 0DH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 9BH, ↙
    ↳ 00H, 00H, 00H, 00H
              else
               %out error: unsupported npad size
               .err
              endif
             endif
            endif
           endif
          endif
         endif
```

```
        endif
       endif
      endif
     endif
    endif
   endif
  endif
 endif
endif
endm
```

.2 ARM

.2.1 Terminologie

ARM a été initialement développé comme un CPU 32-bit, c'est pourquoi ici un *mot*, contrairement au x86, fait 32-bit.

octet 8-bit. La directive d'assemblage DB est utilisée pour définir des variables et des tableaux d'octets.

demi-mot 16-bit. directive d'assemblage DCW —”—.

mot 32-bit. directive d'assemblage DCW —”—.

double mot 64-bit.

quadruple mot 128-bit.

.2.2 Versions

- ARMv4: Le mode Thumb mode a été introduit.
- ARMv6: Utilisé dans la 1ère génération d'iPhone, iPhone 3G (Samsung 32-bit RISC ARM 1176JZ(F)-S qui supporte Thumb-2)
- ARMv7: Thumb-2 a été ajouté (2003). Utilisé dans l'iPhone 3GS, iPhone 4, iPad 1ère génération (ARM Cortex-A8), iPad 2 (Cortex-A9), iPad 3ème génération.
- ARMv7s: De nouvelles instructions ont été ajoutées. Utilisé dans l'iPhone 5, l'iPhone 5c, l'iPad 4ème génération. (Apple A6).
- ARMv8: 64-bit CPU, AKA ARM64 AKA AArch64. Utilisé dans l'iPhone 5S, l'iPad Air (Apple A7). Il n'y a pas de mode Thumb en mode 64-bit, seulement ARM (instructions de 4 octets).

.2.3 ARM 32-bit (AArch32)

Registres d'usage général

- R0 — le résultat d'une fonction est en général renvoyé dans R0
- R1...R12 — GPRs

- R13 — AKA SP (pointeur de pile)
- R14 — AKA LR (link register)
- R15 — AKA PC (program counter)

R0-R3 sont aussi appelés «registres scratch» : les arguments de la fonctions sont d'habitude passés par eux, et leurs valeurs n'ont pas besoin d'être restaurées en sortant de la fonction.

Current Program Status Register (CPSR)

Bit	Description
0..4	M — processor mode
5	T — Thumb state
6	F — FIQ disable
7	I — IRQ disable
8	A — imprecise data abort disable
9	E — data endianness
10..15, 25, 26	IT — if-then state
16..19	GE — greater-than-or-equal-to
20..23	DNM — do not modify
24	J — Java state
27	Q — sticky overflow
28	V — overflow
29	C — carry/borrow/extend
30	Z — zero bit
31	N — negative/less than

Registres VFP (virgule flottante) et registres NEON

<table>
<tr><td>0..31^{bits}</td><td>32..64</td><td>65..96</td><td>97..127</td></tr>
<tr><td colspan="4">Q0$^{128 bits}$</td></tr>
<tr><td colspan="2">D0$^{64 bits}$</td><td colspan="2">D1</td></tr>
<tr><td>S0$^{32 bits}$</td><td>S1</td><td>S2</td><td>S3</td></tr>
</table>

Les registres-S sont 32-bit, utilisés pour le stockage de nombre en simple précision. Les registres-D sont 64-bit, utilisés pour le stockage de nombre en double précision.

Les registres-D et -S partagent le même espace physique dans le CPU—il est possible d'accéder un registre-D via les registres-S (mais c'est insensé).

De même, les registres NEON sont des 128-bit et partagent le même espace physique dans le CPU avec les autres registres en virgule flottante.

En VFP les registres-S sont présents: S0..S31.

En VFPv2 16 registres-D ont été ajoutés, qui occupent en fait le même espace que S0..S31.

En VFPv3 (NEON ou «SIMD avancé») il y a 16 registres-D de plus, D0..D31, mais les registres D16..D31 ne partagent pas l'espace avec aucun autre registres-S.

En NEON ou «SIMD avancé» 16 autres registres-Q 128-bit ont été ajoutés, qui partagent le même espace que D0..D31.

.2.4 ARM 64-bit (AArch64)

Registres d'usage général

Le nombre de registres a été doublé depuis AArch32.

- X0 — le résultat d'une fonction est en général renvoyé dans X0
- X0...X7 — Les arguments de fonction sont passés ici
- X8
- X9...X15 — sont des registres temporaires, la fonction appelée peut les utiliser sans en restaurer le contenu.
- X16
- X17
- X18
- X19...X29 — la fonction appelée peut les utiliser mais doit restaurer leurs valeurs à sa sortie.
- X29 — utilisé comme FP (au moins dans GCC)
- X30 — «Procedure Link Register» AKA LR (link register).
- X31 — ce registre contient toujours zéro AKA XZR ou ZR «Zero Register». Sa partie 32-bit est appelée WZR.
- SP, n'est plus un registre d'usage général.

Voir aussi: [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][11].

La partie 32-bit de chaque registre-X est aussi accessible par les registres-W (W0, W1, etc.).

Partie 32 bits haute	Partie 32 bits basse
X0	
	W0

.2.5 Instructions

Il il y a un suffixe *-S* pour certaines instructions en ARM, indiquant que l'instruction met les flags en fonction du résultat. Les instructions qui n'ont pas ce suffixe ne modifient pas les flags. Par exemple ADD contrairement à ADDS ajoute deux nombres, mais les flags sont inchangés. De telles instructions sont pratiques à utiliser entre CMP où les flags sont mis et, e.g. les sauts conditionnels, où les flags sont utilisés. Elles sont aussi meilleures en termes d'analyse de dépendance de données (car moins de registres sont modifiés pendant leurs exécution).

[11] Aussi disponible en http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

Table des codes conditionnels

Code	Description	Flags
EQ	Égal	Z == 1
NE	Non égal	Z == 0
CS AKA HS (Higher or Same)	Retenue mise/ Non-signé, Plus grand que, égal	C == 1
CC AKA LO (LOwer)	Retenue à zéro / non-signé, moins que	C == 0
MI	moins, négatif / moins que	N == 1
PL	plus, positif ou zéro / Plus grnad que, égal	N == 0
VS	débordement	V == 1
VC	Pas de débordement	V == 0
HI	non signé supérieur / plus grand que	C == 1 et Z == 0
LS	non signé inférieur ou égal / inférieur ou égal	C == 0 ou Z == 1
GE	signé supérieur ou égal / supérieur ou égal	N == V
LT	signé plus petit que / plus petit que	N != V
GT	signé plus grand que / plus grnad que	Z == 0 et N == V
LE	signé inférieur ou égal / moins que, égal	Z == 1 ou N != V
None / AL	toujours	n'importe lequel

.3 MIPS

.3.1 Registres

(Convention d'appel O32)

Registres à usage général GPR

Numéro	Pseudonom	Description
$0	$ZERO	Toujours à zéro. Écrire dans ce registre est comme NOP.
$1	$AT	Utilisé comme un registre temporaire pour les macros assembleurs et les pseudo-instructions.
$2 ...$3	$V0 ...$V1	Le résultat des fonctions est renvoyé par ce registre.
$4 ...$7	$A0 ...$A3	Arguments de fonctions.
$8 ...$15	$T0 ...$T7	Utilisé pour des données temporaires.
$16 ...$23	$S0 ...$S7	Utilisé pour des données temporaireu*.
$24 ...$25	$T8 ...$T9	Utilisé pour des données temporaires.
$26 ...$27	$K0 ...$K1	Réservé pour le noyau de l'OS.
$28	$GP	Pointeur global**.
$29	$SP	SP*.
$30	$FP	FP*.
$31	$RA	RA.
n/a	PC	PC.
n/a	HI	Partie 32-bit haute d'une multiplication ou du reste d'une division***.
n/a	LO	Partie 32-bit basse d'une multiplication ou du reste d'une division***.

Registres en virgule flottante

Nom	Description
$F0..$F1	Le résultat d'une est renvoyé ici.
$F2..$F3	Non utilisé.
$F4..$F11	Utilisé pour des données temporaires.
$F12..$F15	Deux premiers arguments de fonction.
$F16..$F19	Utilisé pour des données temporaires.
$F20..$F31	Utilisé pour des données temporaires.*.

* — L'appelée doit préservé le contenu.
** — L'appelée doit préserver le contenu (sauf dans du code PIC).
*** — Accessible en utilisant les instructions MFHI et MFLO.

.3.2 Instructions

Il y a 3 types d'instructions:

- type-R: celles qui ont 3 registres, Les instructions-R ont habituellement la forme suivante:

  ```
  instruction destination, source1, source2
  ```

  Une chose importante à garder à l'esprit est que lorsque le premier et le second registre sont les même, IDA peut montrer l'instruction sous une forme plus courte:

  ```
  instruction destination/source1, source2
  ```

  Cela nous rappelle quelque peu la syntaxe Intel pour le langage d'assemblage x86.

- type-I: celles qui ont 2 registres et une valeur immédiate 16-bit.
- type-J: instructions de saut/branchement, elles ont 26 bits pour encoder l'offset.

Instructions da saut

Quelle est la différence entre les instructions -B (BEQ, B, etc.) et le -J (JAL, JALR, etc.)?

Les instructions-B ont un type-I, ainsi, l'offset de l'instruction-B est encodé comme une valeur 16-bit immédiate. JR et JALR sont des types-R et sautent à une adresse absolue spécifiée dans un registre. J et JAL sont des type-J, ainsi l'offset est encodé en une valeur 26-bit immédiate.

En bref, les instructions-B peuvent encoder une condition (B est en fait une pseudo-instruction pour `BEQ $ZERO, $ZERO, LABEL`), tandis que les instructions-J ne le peuvent pas.

.4 Quelques fonctions de la bibliothèque de GCC

nom	signification
__divdi3	division signée
__moddi3	reste (modulo) d'une division signée
__udivdi3	division non signée
__umoddi3	reste (modulo) d'une division non signée

.5 Quelques fonctions de la bibliothèque MSVC

ll dans une fontion signifie «long long», i.e., type de donées 64-bit.

nom	signification
__alldiv	division signée
__allmul	multiplication
__allrem	reste de la division signée
__allshl	décalage à gauche
__allshr	décalage signé à droite
__aulldiv	division non signée
__aullrem	reste de la division non signée
__aullshr	décalage non signé à droite

La multiplication et le décalage à gauche sont similaire pour les nombres signés et non signés, donc il n'y a qu'une seule fonction ici.

Le code source des ces fonctions peut être trouvé dans l'installation de MSVS, dans VC/crt/src/intel/*.asm.

.6 Cheatsheets

.6.1 IDA

Anti-sèche des touches de raccourci:

touche	signification
Space	échanger le listing et le mode graphique
C	convertir en code
D	convertir en données
A	convertir en chaîne
*	convertir en tableau
U	rendre indéfini
O	donner l'offset d'une opérande
H	transformer en nombre décimal
R	transformer en caractère
B	transformer en nombre binaire
Q	transformer en nombre hexa-décimal
N	renommer l'identifiant
?	calculatrice
G	sauter à l'adresse
:	ajouter un commentaire
Ctrl-X	montrer les références à la fonction, au label, à la variable courant inclure dans la pile locale
X	montrer les références à la fonction, au label, à la variable, etc.
Alt-I	chercher une constante
Ctrl-I	chercher la prochaine occurrence d'une constante
Alt-B	chercher une séquence d'octets
Ctrl-B	chercher l'occurrence suivante d'une séquence d'octets
Alt-T	chercher du texte (instructions incluses, etc.)
Ctrl-T	chercher l'occurrence suivante du texte
Alt-P	éditer la fonction courante
Enter	sauter à la fonction, la variable, etc.
Esc	retourner en arrière
Num -	cacher/plier la fonction ou la partie sélectionnée
Num +	afficher la fonction ou une partie

cacher une fonction ou une partie de code peut être utile pour cacher des parties du code lorsque vous avez compris ce qu'elles font. ceci est utilisé dans mon script[12]pour cacher des patterns de code inline souvent utilisés.

.6.2 OllyDbg

Anti-sèche des touches de raccourci:

raccourci	signification
F7	tracer dans la fonction
F8	enjamber
F9	démarrer
Ctrl-F2	redémarrer

.6.3 MSVC

. . Quelques options utiles qui ont été utilisées dans ce livre

[12] GitHub

option	signification
/O1	minimiser l'espace
/Ob0	pas de mire en ligne
/Ox	optimisation maximale
/GS-	désactiver les vérifications de sécurité (buffer overflows)
/Fa(file)	générer un listing assembleur
/Zi	activer les informations de débogage
/Zp(n)	aligner les structures sur une limite de n-octet
/MD	l'exécutable généré utilisera MSVCR*.DLL

Quelques informations sur les versions de MSVC: 5.1.1 on page 908.

.6.4 GCC

Quelques options utiles qui ont été utilisées dans ce livre.

option	signification
-Os	optimiser la taille du code
-O3	optimisation maximale
-regparm=	nombre d'arguments devant être passés dans les registres
-o file	définir le nom du fichier de sortie
-g	mettre l'information de débogage dans l'exécutable généré
-S	générer un fichier assembleur
-masm=intel	construire le code source en syntaxe Intel
-fno-inline	ne pas mettre les fonctions en ligne

.6.5 GDB

Quelques commandes que nous avons utilisées dans ce livre:

option	
break filename.c:number	mettre un point d'arrêt à la ligne number du code source
break function	mettre un point d'arrêt sur une fonction
break *address	mettre un point d'arrêt à une adresse
b	—"—
p variable	afficher le contenu d'une variable
run	démarrer
r	—"—
cont	continuer l'exécution
c	—"—
bt	afficher la pile
set disassembly-flavor intel	utiliser la syntaxe Intel
disas	disassemble current function
disas function	désassembler la fonction
disas function,+50	disassemble portion
disas $eip,+0x10	—"—
disas/r	désassembler avec les opcodes
info registers	afficher tous les registres
info float	afficher les registres FPU
info locals	afficher les variables locales
x/w ...	afficher la mémoire en mot de 32-bit
x/w $rdi	afficher la mémoire en mot de 32-bit à l'adresse dans RDI
x/10w ...	afficher 10 mots de la mémoire
x/s ...	afficher la mémoire en tant que chaîne
x/i ...	afficher la mémoire en tant que code
x/10c ...	afficher 10 caractères
x/b ...	afficher des octets
x/h ...	afficher en demi-mots de 16-bit
x/g ...	afficher des mots géants (64-bit)
finish	exécuter jusqu'à la fin de la fonction
next	instruction suivante (ne pas descendre dans les fonctions)
step	instruction suivante (descendre dans les fonctions)
set step-mode on	ne pas utiliser l'information du numéro de ligne en exécutant pas à pas
frame n	échanger la stack frame
info break	afficher les points d'arrêt
del n	effacer un point d'arrêt
set args ...	définir les arguments de la ligne de commande

Acronymes utilisés

OS Système d'exploitation (Operating System) xix

POO Programmation orientée objet 700

LP Langage de programmation xvi

PRNG Nombre généré pseudo-aléatoirement x

ROM Mémoire morte 111

UAL Unité arithmétique et logique 36

PID ID d'un processus 1054

LF Line feed (10 ou '\n' en C/C++) 675

CR Carriage return (13 ou '\r' en C/C++) 675

LIFO Dernier entré, premier sorti 42

MSB Bit le plus significatif 407

LSB Bit le moins significatif

NSA National Security Agency (Agence Nationale de la Sécurité) 1030

CFB Cipher Feedback 1119

CSPRNG Cryptographically Secure Pseudorandom Number Generator (générateur de nombres pseudo-aléatoire cryptographiquement sûr) 1120

ABI Application Binary Interface 22

PC Program Counter. IP/EIP/RIP dans x86/64. PC dans ARM. 26

SP pointeur de pile. SP/ESP/RSP dans x86/x64. SP dans ARM. 26

RA Adresse de retour 30

PE Portable Executable 7

DLL Dynamic-Link Library 985

LR Link Register 9

IDA Désassembleur interactif et débogueur développé par Hex-Rays 7

IAT Import Address Table 986

INT Import Name Table 986

RVA Relative Virtual Address 986

VA Virtual Address 986

OEP Original Entry Point 971

MSVC Microsoft Visual C++

MSVS Microsoft Visual Studio 1341

ASLR Address Space Layout Randomization 789

MFC Microsoft Foundation Classes 991

TLS Thread Local Storage 361

AKA Also Known As — Aussi connu sous le nom de 42

CRT C Runtime library 14

CPU Central Processing Unit xix

GPU Graphics Processing Unit 1134

FPU Floating-Point Unit v

CISC Complex Instruction Set Computing . . . 27

RISC Reduced Instruction Set Computing . . . 3

GUI Graphical User Interface . . . 981

RTTI Run-Time Type Information . . . 720

BSS Block Started by Symbol . . . 35

SIMD Single Instruction, Multiple Data . . . 256

BSOD Blue Screen of Death . . . 972

DBMS Database Management Systems . . . 585

ISA Instruction Set Architecture . . . xi

HPC High-Performance Computing . . . 664

SEH Structured Exception Handling . . . 52

ELF Executable and Linkable Format: Format de fichier exécutable couramment utilisé sur les systèmes *NIX, Linux inclus . . . 109

TIB Thread Information Block . . . 361

PIC Position Independent Code . . . 693

NAN Not a Number . . . 1328

NOP No Operation . . . 9

BEQ (PowerPC, ARM) Branch if Equal . . . 129

BNE (PowerPC, ARM) Branch if Not Equal . . . 273

BLR (PowerPC) Branch to Link Register . . . 1083

XOR eXclusive OR (OU exclusif) 1337

MCU Microcontroller Unit 634

RAM Random-Access Memory 4

GCC GNU Compiler Collection 5

EGA Enhanced Graphics Adapter 1298

VGA Video Graphics Array 1298

API Application Programming Interface 805

ASCII American Standard Code for Information Interchange 373

ASCIIZ ASCII Zero (chaîne ASCII terminée par un octet nul (à zéro)) 126

IA64 Intel Architecture 64 (Itanium) 588

EPIC Explicitly Parallel Instruction Computing 1294

OOE Out-of-Order Execution 589

MSDN Microsoft Developer Network 794

STL (C++) Standard Template Library 729

PODT (C++) Plain Old Data Type 745

VM Virtual Memory (mémoire virtuelle)

WRK Windows Research Kernel 932

GPR General Purpose Registers 2

SSDT System Service Dispatch Table 973

TOS Top of Stack 857

LVA (Java) Local Variable Array 865

JVM Java Virtual Machine x

JIT Just-In-Time compilation 855

CDFS Compact Disc File System 939

CD Compact Disc

ADC Analog-to-Digital Converter 935

EOF End of File (fin de fichier) 118

DIY Do It Yourself 799

MMU Memory Management Unit 786

DES Data Encryption Standard 579

MIME Multipurpose Internet Mail Extensions 579

DBI Dynamic Binary Instrumentation 672

XML Extensible Markup Language 811

JSON JavaScript Object Notation 811

URL Uniform Resource Locator 6

IV Initialization Vector xii

RSA Rivest Shamir Adleman 1237

CPRNG Cryptographically secure PseudoRandom Number Generator 1238

GiB Gibibyte 1255

CRC Cyclic redundancy check 1278

AES Advanced Encryption Standard 1278

GC Garbage Collector 796

IDE Integrated development environment 485

BB Basic Block 1312

Glossaire

anti-pattern En général considéré comme une mauvaise pratique. 45, 105, 588

atomic operation « $\alpha\tau o\mu o\varsigma$ » signifie «indivisible » en grec, donc il est garantie qu'une opération atomique ne sera pas interrompue par d'autres threads. 826, 1027

basic block un groupe d'instructions qui n'a pas d'instruction de saut/branchement, et donc n'as pas de saut de l'intérieur du bloc vers l'extérieur. Dans IDA il ressemble à une liste d'instructions sans ligne vide. 896, 1298

callee Une fonction appelée par une autre. 46, 65, 92, 118, 132, 135, 138, 544, 588, 702, 841, 953, 955–957, 961, 962, 1353

caller Une fonction en appelant une autre. 8–11, 14, 41, 65, 118, 132–134, 137, 147, 206, 544, 601, 702, 953, 954, 956, 957, 962

compiler intrinsic Une foncion spécifique d'un compilateur, qui n'est pas une fonction usuelle de bibliothèque. Le compilateur génère du code machine spécifique au lieu d'un appel à celui-ci. Souvent il s'agit d'une pseudo-fonction pour une instruction CPU spécifique. Lire plus: (11.4 on page 1290). 1341

CP/M Control Program for Microcomputers: un OS de disque très basique utilisé avant MS-DOS. 1179

dongle Un dongle est un petit périphérique se connectant sur un port d'imprimante LPT (par le passé) ou USB. Sa fonction est similaire au tokens de sécurité, il a de la mémoire et, parfois, un algorithme secret de (crypto-)hachage.. 1081

décrémenter Décrémenter de 1. 26, 242, 266, 569, 942, 1331, 1335, 1341

endianness Ordre des octets: 2.2 on page 586. 30, 108, 1337

GiB Gibioctet: 2^{30} or 1024 mebioctets ou 1073741824 octets. 22

incrémenter Incrémenter de 1. 23, 27, 242, 247, 266, 272, 420, 423, 569, 1331

jump offset une partie de l'opcode de l'instruction JMP ou Jcc, qui doit être ajoutée à l'adresse de l'instruction suivante, et c'est ainsi que le nouveau PC est calculé. Peut être négatif. 127, 174, 175, 1331

kernel mode Un mode CPU sans restriction dans lequel le noyau de l'OS et les drivers sont exécutés. cf. user mode. 1362

leaf function Une fonction qui n'appelle pas d'autre fonction. 39, 45

link register (RISC) Un registre où l'adresse de retour est en général stockée. Ceci permet d'appeler une fonction leaf sans utiliser la pile, i.e, plus rapidemment. 45, 1083, 1349, 1351

loop unwinding C'est lorsqu'un compilateur, au lieu de générer du code pour une boucle de n itérations, génère juste n copies du corps de la boucle, afin de supprimer les instructions pour la gestion de la boucle. 245

moyenne arithmétique la somme de toutes les valeurs, divisé par leur nombre. 666

name mangling utilisé au moins en C++, où le compilateur doit encoder le nom de la classe, la méthode et le type des arguments dans une chaîne, qui devient le nom interne de la fonction. Vous pouvez en lire plus à ce propos ici: 3.21.1 on page 699. 699, 909

NaN pas un nombre: un cas particulier pour les nombres à virgule flottante, indiquant généralement une erreur. 304, 326, 327, 1297

NEON AKA «Advanced SIMD» — SIMD de ARM. 1350

nombre réel nombre qui peut contenir un point. comme *float* et *double* en C/C++. 285

NOP «no operation», instruction ne faisant rien. 942

NTAPI API disponible seulement dans la série de Windows NT. Très peu documentée par Microsoft. 1038

PDB (Win32) Fichier contenant des informations de débogage, en général seulement les noms des fonctions, mais aussi parfois les arguments des fonctions et le nom des variables locales. 908, 989, 1038, 1040, 1047, 1048, 1156

pointeur de pile Un registre qui pointe dans la pile. 13, 15, 27, 42, 43, 49, 60, 75, 77, 101, 135, 702, 841, 953, 955–957, 1325, 1332, 1349

POKE instruction du langage BASIC pour écrire un octet à une adresse spécifique. 943

produit Résultat d'une multiplication. 133, 292, 296, 527, 559, 1287, 1288

quotient Résultat de la division. 285, 288, 290, 291, 296, 558, 636, 668

register allocator La partie du compilateur qui assigne des registes du CPU aux variables locales. 264, 395, 545

reverse engineering action d'examiner et de comprendre comment quelque chose fonctionne, parfois dans le but de le reproduire. iv, 1340

security cookie Une valeur aléatoire, différente à chaque exécution. Vous pouvez en lire plus à ce propos ici: 1.26.3 on page 358. 1015

stack frame Une partie de la pile qui contient des informations spécifiques à la fonction courante: variables locales, arguments de la fonction, RA, etc.. 94, 133, 610, 611, 1015

stdout standard output, sortie standard. 29, 50, 206

tail call C'est lorsque le compilateur (ou l'interpréteur) transforme la récursion (ce qui est possible: *tail recursion*) en une itération pour l'efficacité: wikipedia. 615

tas Généralement c'est un gros bout de mémoire fournit par l'OS et utilisé par les applications pour le diviser comme elles le souhaitent. malloc()/free() fonctionnent en utilisant le tas. 43, 447, 724, 727, 728, 745, 747, 768, 769, 812, 984, 986

thunk function Minuscule fonction qui a un seul rôle: appeler une autre fonction. 31, 59, 504, 1083, 1095

tracer Mon propre outil de debugging. Vous pouvez en lire plus à son propos ici: 7.2.1 on page 1031. 248–250, 697, 793, 913, 928, 932, 933, 1009, 1075–1077, 1159, 1167, 1172, 1173, 1176, 1192, 1289

type de donnée intégral nombre usuel, mais pas un réel. peut être utilisé pour passer des variables de type booléen et des énumérations. 301

user mode Un mode CPU restreint dans lequel le code de toutes les applications est exécuté. cf. kernel mode. 1105, 1361

Windows NT Windows NT, 2000, XP, Vista, 7, 8, 10. 373, 541, 840, 918, 973, 987, 1026, 1183, 1340

word Dans les ordinateurs plus vieux que les PCs, la taille de la mémoire était souvent mesurée en mots plutôt qu'en octet. 577, 579–582, 732, 813

xoring souvent utilisé en anglais, qui signifie appliquer l'opération XOR. 1015, 1098, 1103

Index